PM5358

S/UNI-4x622

DRIVER MANUAL

PROPRIETARY AND CONFIDENTIAL

PRELIMINARY

ISSUE 1: APRIL, 2001

ABOUT THIS MANUAL AND S/UNI-4X622

This manual describes the S/UNI-4x622 (PM5358) device driver. It describes the driver's functions, data structures, and architecture. This manual focuses on the driver's interfaces and their relationship to your application, real-time operating system, and to the device. It also describes in general terms how to modify and port the driver to your software and hardware platform.

Audience

This manual was written for people who need to:

- Evaluate and test the S/UNI-4x622 devices
- Modify and add to the S/UNI-4x622 driver's functions
- Port the S/UNI-4x622 driver to a particular platform.

References

For more information about the S/UNI-4x622 driver, see the driver's release notes. For more information about the S/UNI-4x622 device, see the documents listed in Table 1 and any related errata documents.

Table 1: Related Documents

Document Number	Document Name
PMC-1991044	Saturn User Network Interface (4x622) Telecom Standard Product Data Sheet

Note: Ensure that you use the document that PMC-Sierra issued for your version of the device and driver.

Revision History

Issue No.	Issue Date	Details of Change
Issue 1	April, 2001	Document created

Legal Issues

PMC-2010419 (P1), ref PMC-2000459 (P2)

Contacting PMC-Sierra

PMC-Sierra
8555 Baxter Place Burnaby, BC
Canada V5A 4V7

Tel: (604) 415-6000
Fax: (604) 415-6200

Document Information: document@pmc-sierra.com
Corporate Information: info@pmc-sierra.com
Technical Support: apps@pmc-sierra.com
Web Site: http://www.pmc-sierra.com

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The following sections of the S/UNI-4x622 Device Driver Design Specification describe the S/UNI-4x622 device driver. The code provided throughout this document is written in ANSI-C. This has been done to promote greater driver portability to other embedded hardware (Section 5) and Real-Time Operating System (RTOS) environments (Section 6).

Section 2 of this document, Software Architecture, defines the software architecture of the S/UNI-4x622 device driver by including a discussion of the driver's external interfaces and its main components. The Data Structure information in Section 3 describes the elements of the driver that either configure or control its behavior. Included here are the constants, variables, and structures that the S/UNI-4x622 device driver uses to store initialization, configuration, and status information. Section 4 provides a detailed description of each function that is a member of the S/UNI-4x622 driver Application Programming Interface (API). This section outlines function calls that hide device-specific details and application callbacks that notify the user of significant device events.

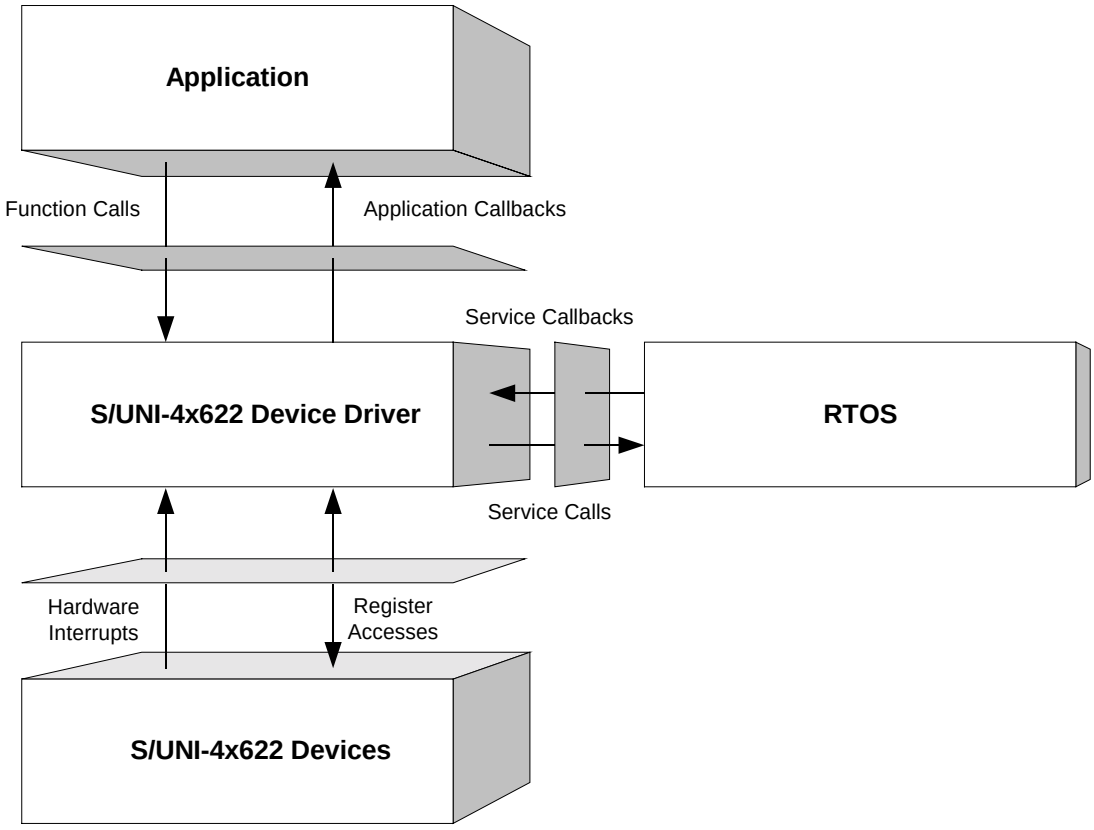
2 SOFTWARE ARCHITECTURE

This section describes the software architecture of the S/UNI-4x622 device driver. This includes a discussion of the driver's external interfaces and its main components.

2.1 Driver External Interfaces

Figure 1 illustrates the external interfaces defined for the S/UNI-4x622 device driver.

Figure 1: Driver External Interfaces



Application Programming Interface

The driver Application Programming Interface (API) is a list of high-level functions that can be invoked by application programmers to configure, control and monitor S/UNI-4x622 devices. The API functions perform operations that are more meaningful from a system's perspective. The API includes functions such as:

- Initialize the device(s)
- Perform diagnostic tests
- Validate configuration information
- Retrieve status and counts information

The driver API functions use the services of the other driver components to provide this system-level functionality to the application programmer.

The driver API also consists of callback routines that are used to notify the application of significant events that take place within the device(s) and module.

Real-Time OS Interface

The driver's RTOS interface provides functions that let the driver use RTOS services. The driver requires the memory, interrupt, and preemption services from the RTOS. The RTOS interface functions perform the following tasks for the driver:

- Allocate and de-allocate memory
- Manage buffers for the ISR and the DPR
- Enable and disable preemption

The RTOS interface also includes service callbacks. These are functions installed by the driver using RTOS service calls such as installing interrupts. These service callbacks are invoked when an interrupt occurs.

Note: You must modify RTOS interface code to suit your RTOS.

Hardware Interface

The hardware interface provides functions that read from and write to the device registers. The hardware interface also provides a template for an ISR that the driver calls when the device raises a hardware interrupt. You must modify this function based on the interrupt configuration of your system.

2.2 Main Components

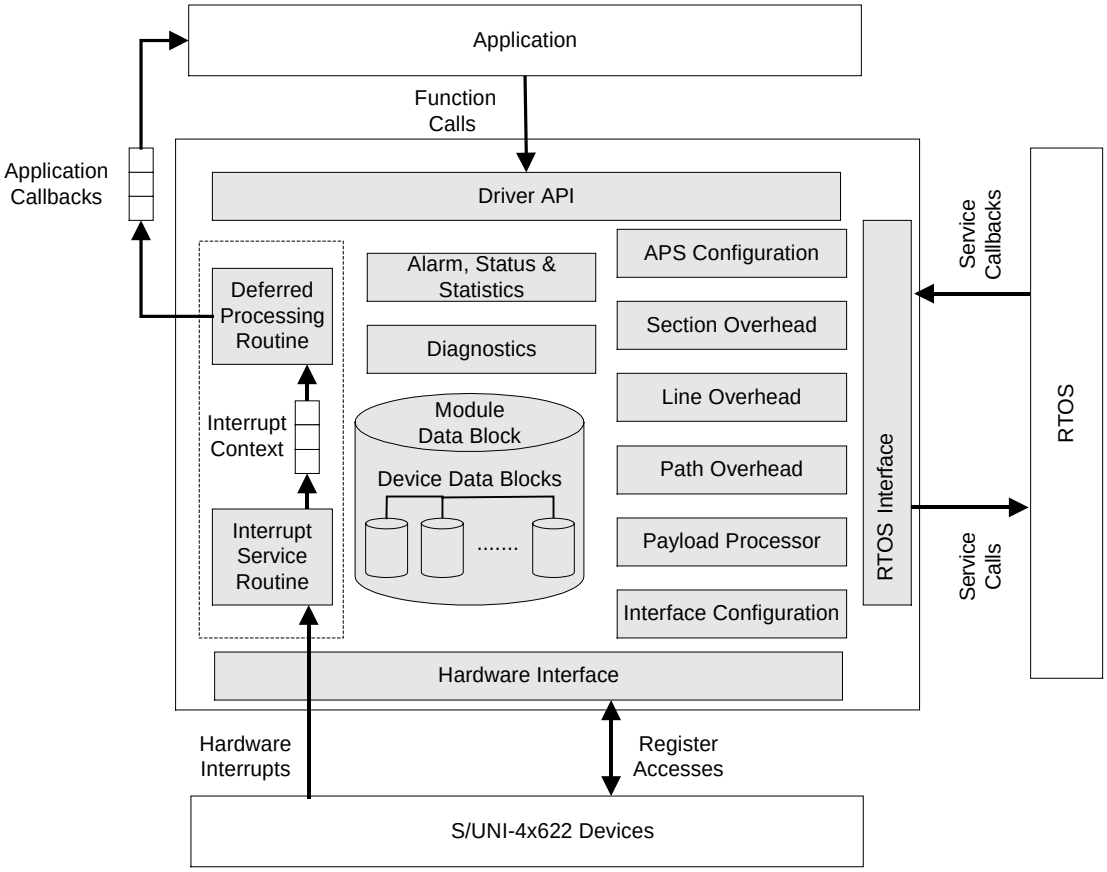
Figure 2 illustrates the top level architectural components of the S/UNI-4x622 device driver. This applies in both polled and interrupt driven operation. In polled operation the ISR is called periodically. In interrupt operation the interrupt directly triggers the ISR.

The driver includes eight main components:

- Module and device(s) data-blocks
- Interrupt-service routine
- Deferred-processing routine

- Alarm, status and counts
- Section Overhead
- Line Overhead
- Path Overhead
- Payload Processor
- Interface Configuration
- APS Configuration

Figure 2: Driver Architecture



Module Data-Block and Device(s) Data-Blocks

The Module Data-Block (MDB) is the top layer data structure, created by the S/UNI-4x622 driver to store context information about the driver module, such as:

- Module state

- Maximum number of devices
- The DDB(s)

The Device Data-Block (DDB) is contained in the MDB, and initialized by the driver module for each S/UNI-4x622 device that is registered. There is one DDB per device and there is a limit on the number of DDBs, and that limit is set by the USER when the module is initialized. The DDB is used to store context information about one device, such as:

- Device state
- Control information
- Initialization parameters
- Callback function pointers

Interrupt-Service Routine

The S/UNI-4x622 driver provides an ISR called `suni4x622ISR` that checks if there is any valid interrupt condition present for the device. This function can be used by a system-specific interrupt-handler function to service interrupts raised by the device.

The low-level interrupt-handler function that traps the hardware interrupt and calls `suni4x622ISR` is system and RTOS dependent. Therefore, it is outside the scope of the driver. Example implementations of an interrupt handler and functions that install and remove it are provided as a reference in section 5.2. You can customize these example implementations to suit your specific needs.

See section 2.5 for a detailed explanation of the ISR and interrupt-servicing model.

Deferred-Processing Routine

The S/UNI-4x622 driver provides a DPR called `suni4x622DPR` that processes any interrupt condition gathered by the ISR for that device. Typically, a system specific function, which runs as a separate task within the RTOS, will call `suni4x622DPR`.

Example implementations of a DPR task and functions that install and remove it are provided as a reference in section 5.2. You can customize these example implementations to suit your specific needs.

See section 2.5 for a detailed explanation of the DPR and interrupt-servicing model.

Alarms, Status and Counts

The alarm, status and counts section is responsible for monitoring alarms, tracking devices status information and retrieving counts for each device registered with (added to) the driver.

Section Overhead

The Section Overhead section provides functions to control and monitor the section overhead processing. Read / Write access is given to the section trace message (J0). This message is compared with a configurable reference and mismatches are reported. Section BIP-8 (B1) errors are accumulated in a counter that can be read. Section overhead alarms are detected and reported. For diagnostic purposes, errors can be introduced in the section overhead bytes.

Line Overhead

The Line Overhead section provides functions to configure and monitor the line overhead on both the receive and transmit sides. Read / Write access is given to the APS bytes (K1 and K2) and most other overhead bytes. Line BIP-8 (B2) errors are accumulated in a counter that can be read. Line overhead alarms are detected and reported. For diagnostic purposes, errors can be introduced in the line overhead bytes. Additional functions are provided to automatically insert line RDI and line AIS.

Path Overhead

The Path Overhead section provides functions to configure and monitor the path overhead on both the receive and transmit sides. Read / Write access is given to the path trace message (J1) and the path signal label (C2). Both are compared with a configurable reference and mismatches are reported. Path BIP-8 (B3) errors and REI are accumulated in a counter that can be read. Path overhead alarms are detected and reported. For diagnostic purposes, errors can be introduced in the path overhead bytes. Additional functions are provided to automatically insert path AIS, and force generation of individual outgoing justification events.

Payload Processor

The Payload Processor section provides functions to configuring the payload for ATM or POS processing. Function is provided to configure ATM/POS processing.

Interface Configuration

The Interface Configuration section provides functions to configure the FIFO, line and system side interface for ATM or POS mode. Functions are provided for FIFO management to separate the line side timing from the higher layer ATM/POS link layer timing. The Line interface is responsible for receive/transmit line clock configuration. The System interface is responsible for configuring the system to UTOPIA Level 3 or POS-PHY Level 3 interface for either ATM or POS application.

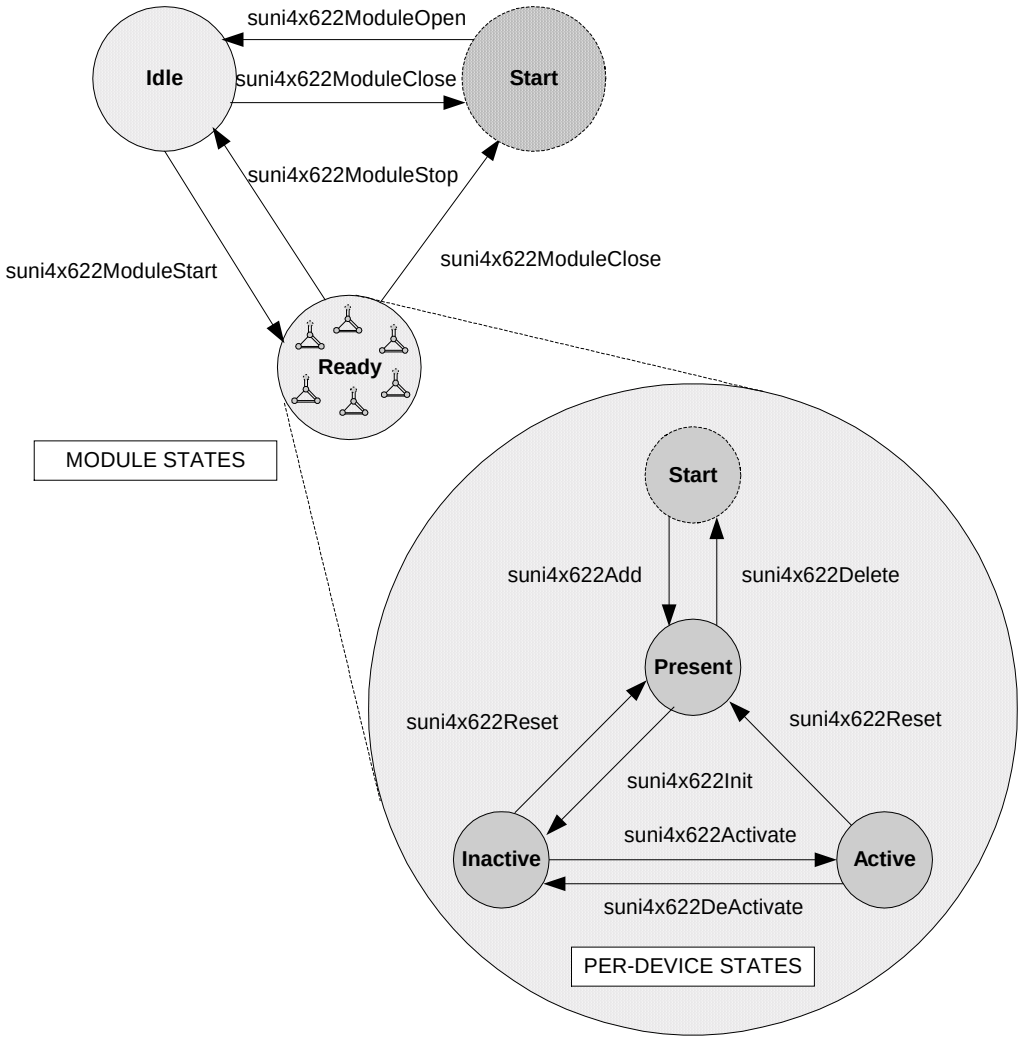
APS Configuration

The APS Configuration section provides function to configure the operating mode for the device to either a protect or working mate in a APS failover condition.

2.3 Software States

Figure 3 shows the software state diagram for the S/UNI-4x622 driver. State transitions occur on the successful execution of the corresponding transition functions shown. State information helps maintain the integrity of the MDB and DDB(s) by controlling the set of operations allowed in each state.

Figure 3: Driver Software States



Module States

The following is a description of the S/UNI-4x622 module states. See section 4.1 for a detailed description of the API functions that are used to change the module state.

Start

The driver module has not been initialized. In this state the driver does not hold any RTOS resources (memory, timers, etc); has no running tasks, and performs no actions.

Idle

The driver module has been initialized successfully. The Module Initialization Vector (MIV) has been validated, the Module Data Block (MDB) has been allocated and loaded with current data, the per-device data structures have been allocated, and the RTOS has responded without error to all the requests sent to it by the driver.

Ready

This is the normal operating state for the driver module. This means that all RTOS resources have been allocated and the driver is ready for devices to be added. The driver module remains in this state while devices are in operation.

Device States

The following is a description of the S/UNI-4x622 per-device states. The state that is mentioned here is the software state as maintained by the driver, and not as maintained inside the device itself. See section 4.3 for a detailed description of the API functions that are used to change the per-device state.

Start

The device has not been initialized. In this state the device is unknown by the driver and performs no actions. There is a separate flow for each device that can be added, and they all start here.

Present

The device has been successfully added. A Device Data Block (DDB) has been associated to the device and updated with the user context, and a device handle has been given to the USER. In this state the device performs no actions.

Inactive

In this state the device is configured but all data functions are de-activated including interrupts and alarms, as well as status and counts functions.

Active

This is the normal operating state for the device. In this state, interrupt servicing or polling is enabled.

2.4 Processing Flows

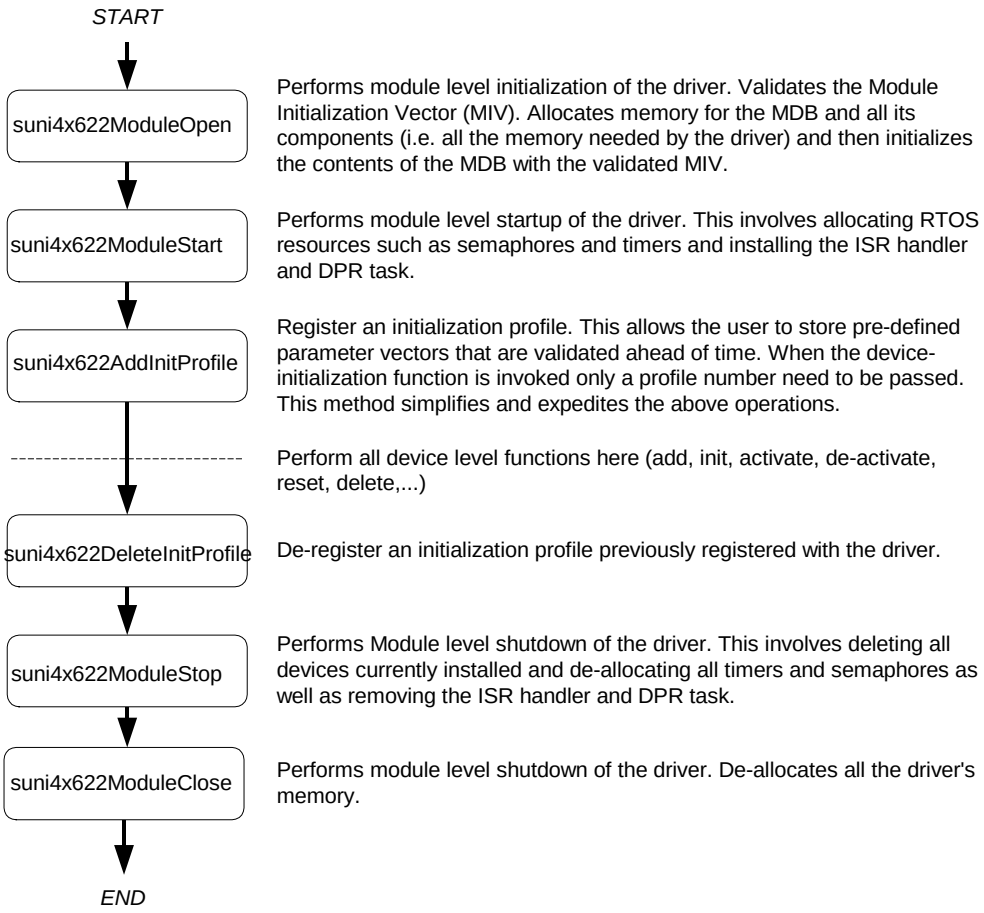
This section describes the main processing flows of the S/UNI-4x622 driver components.

The flow diagrams presented here illustrate the sequence of operations that take place for different driver functions. The diagrams also serve as a guide to the application programmer by illustrating the sequence in which the application must invoke the driver API.

Module Management

The following diagram illustrates the typical function call sequences that occur when initializing or shutting down the S/UNI-4x622 driver module.

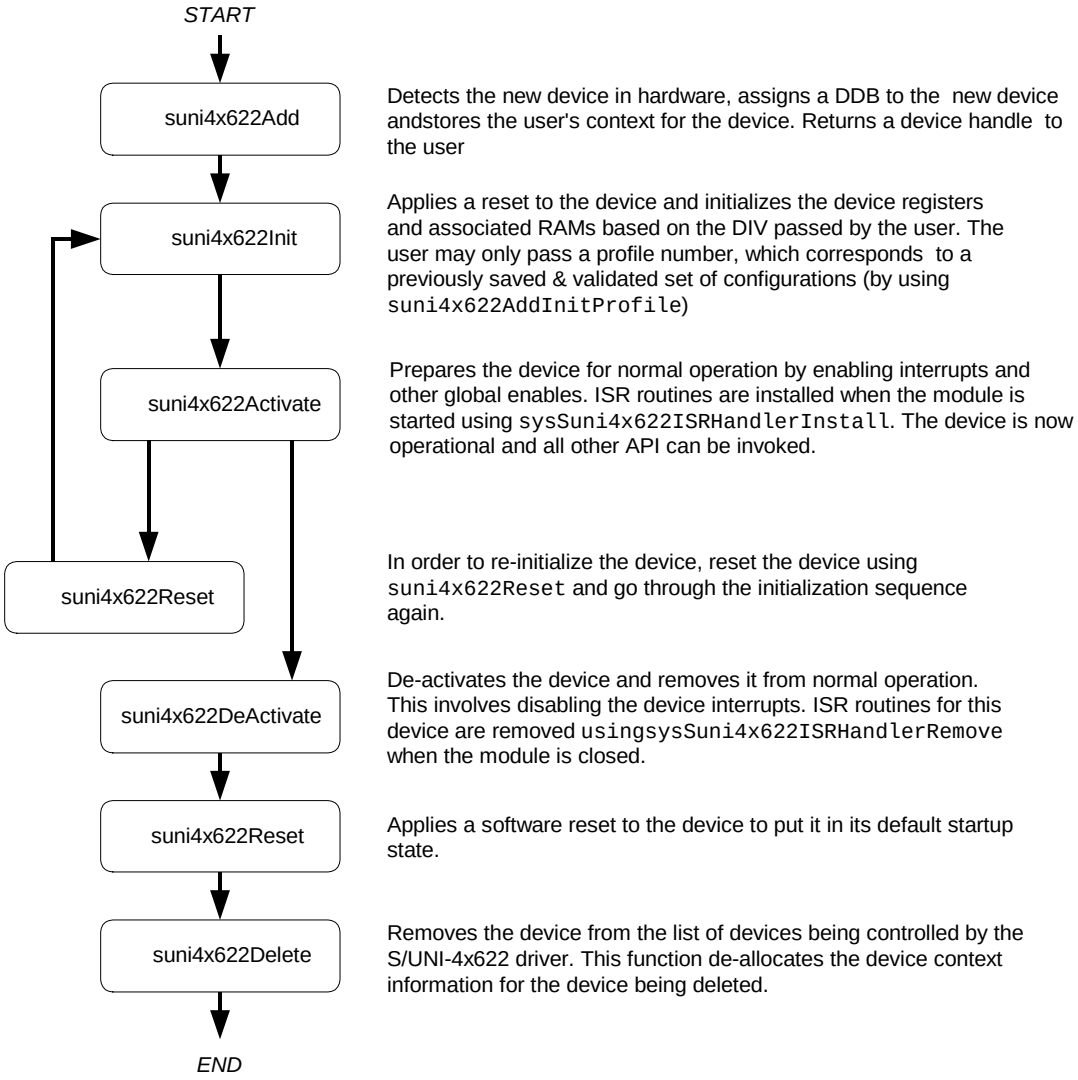
Figure 4: Module Management Flow Diagram



Device Management

The following figure shows the typical function call sequences that the driver uses to add, initialize, re-initialize, and delete the S/UNI-4x622 device.

Figure 5: Device Management Flow Diagram



2.5 Interrupt Servicing

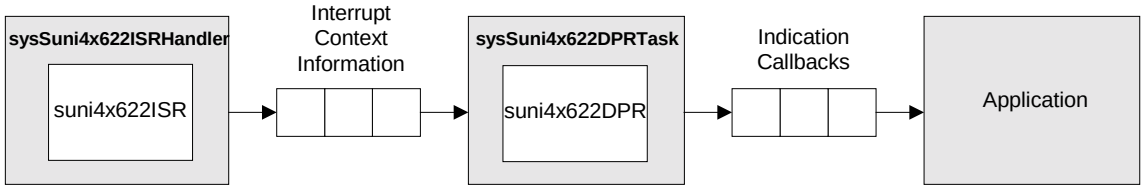
The S/UNI-4x622 driver services device interrupts using an Interrupt-Service Routine (ISR) that traps interrupts, and a Deferred-Processing Routine (DPR) that actually processes the interrupt conditions and clears them. This lets the ISR execute quickly and exit. Most of the time-consuming processing of the interrupt conditions is deferred to the DPR by queuing the necessary interrupt-context information to the DPR task. The DPR function runs in the context of a separate task within the RTOS.

Note: Since the DPR task processes potentially serious interrupt conditions, you should set the DPR task's priority higher than the application task interacting with the S/UNI-4x622 driver.

The driver provides system-independent functions, `suni4x622ISR` and `suni4x622DPR`. You must fill in the corresponding system-specific functions, `sysSuni4x622ISRHandler` and `sysSuni4x622DPRTask`. The system-specific functions isolate the system-specific communication mechanism (between the ISR and DPR) from the system-independent functions, `suni4x622ISR` and `suni4x622DPR`.

Figure 6 illustrates the interrupt service model used in the S/UNI-4x622 driver design.

Figure 6: Interrupt Service Mode



Note: Instead of using an interrupt service model, you can use a polling service model in the S/UNI-4x622 driver to process the device's event-indication registers (see page 26).

Calling suni4x622ISR

An interrupt handler function, which is system dependent, must call `suni4x622ISR`. But first, the low-level interrupt-handler function must trap the device interrupts. You must implement this function to fit your own system. As a reference, an example implementation of the interrupt handler (`sysSuni4x622ISRHandler`) appears on page 110. You can customize this example implementation to suit your needs.

The interrupt handler that you implement (`sysSuni4x622ISRHandler`) is installed in the interrupt vector table of the system processor. It is called when one or more S/UNI-4x622 devices interrupt the processor. The interrupt handler then calls `suni4x622ISR` for each device in the active state that has interrupt processing enabled.

The `suni4x622ISR` function reads from the master interrupt-status registers and the miscellaneous interrupt-status registers of the S/UNI-4x622. If at least one valid interrupt condition is found then `suni4x622ISR` fills an Interrupt-Service Vector (ISV) with this status information as well as the current device handle. The `suni4x622ISR` function also clears and disables all the device's interrupts detected. The `sysSuni4x622ISRHandler` function is then responsible to send this ISV buffer to the DPR task.

Note: Normally you should save the status information for deferred processing by implementing a message queue. The interrupt handler sends the status information to the queue by the `sysSuni4x622ISRHandler`.

Calling suni4x622DPR

The sysSuni4x622DPRTask function is a system specific function that runs as a separate task within the RTOS. You should set the DPR task's priority higher than the application task(s) interacting with the S/UNI-4x622 driver. In the message-queue implementation model, this task has an associated message queue. The task waits for messages from the ISR on this message queue. When a message arrives, sysSuni4x622DPRTask calls the DPR (suni4x622DPR) with the received ISV.

Then suni4x622DPR processes the status information and takes appropriate action based on the specific interrupt condition detected. The nature of this processing can differ from system to system. Therefore, suni4x622DPR calls different indication callbacks for different interrupt conditions.

Typically, you should implement these callback functions as simple message posting functions that post messages to an application task. However, you can implement the indication callback to perform processing within the DPR task context and return without sending any messages. In this case, ensure that this callback function does not call any API functions that would change the driver's state, such as suni4x622Delete. Also, ensure that the callback function is non-blocking because the DPR task executes while S/UNI-4x622 interrupts are disabled. You can customize these callbacks to suit your system. See page 103 for example implementations of the callback functions.

Note: Since the suni4x622ISR and suni4x622DPR routines themselves do not specify a communication mechanism, you have full flexibility in choosing a communication mechanism between the two. A convenient way to implement this communication mechanism is to use a message queue, which is a service that most RTOSs provide.

You must implement the two system specific functions, sysSuni4x622ISRHandler and sysSuni4x622DPRTask. When the driver calls sysSuni4x622ISRHandlerInstall, the application installs sysSuni4x622ISRHandler in the interrupt vector table of the processor, and the sysSuni4x622DPRTask function is spawned as a task by the application. The sysSuni4x622ISRHandlerInstall function also creates the communication channel between sysSuni4x622ISRHandler and sysSuni4x622DPRTask. This communication channel is most commonly a message queue associated with the sysSuni4x622DPRTask.

Similarly, during removal of interrupts, the driver removes sysSuni4x622ISRHandler from the microprocessor's interrupt vector table and deletes the task associated with sysSuni4x622DPRTask.

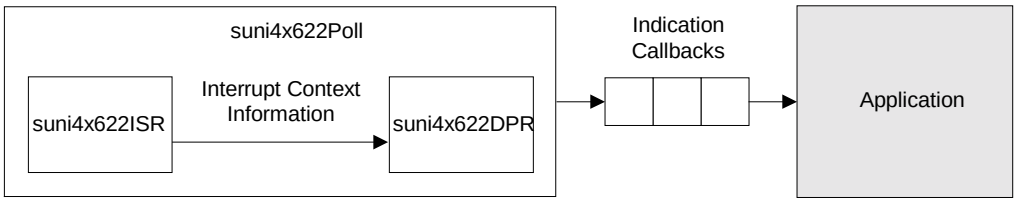
As a reference, this manual provides example implementations of the interrupt installation and removal functions on pages 109 and 111. You can customize these prototypes to suit your specific needs.

Calling suni4x622Poll

Instead of using an interrupt service model, you can use a polling service model in the S/UNI-4x622 driver to process the device's event-indication registers.

Figure 7 illustrates the polling service model used in the S/UNI-4x622 driver design.

Figure 7: Polling Service Model



In polling mode, the application is responsible for calling `suni4x622Poll` often enough to service any pending error or alarm conditions. When `suni4x622Poll` is called, the `suni4x622ISR` function is called internally.

The `suni4x622ISR` function reads from the master interrupt-status registers and the miscellaneous interrupt-status registers of the S/UNI-4x622. If at least one valid interrupt condition is found then `suni4x622ISR` fills an Interrupt-Service Vector (ISV) with this status information as well as the current device handle. In polling mode, this ISV buffer is passed to the DPR task by calling `suni4x622DPR` internally.

3 DATA STRUCTURES

This section describes the elements of the driver that configure or control its behavior, and should therefore be of interest to the application programmer. Included here are the constants, variables and structures that the S/UNI-4x622 device driver uses to store initialization, configuration and counts information. The channel number starts from 0. The structure contains arrays of four elements, where index 0 corresponds to the first channel and index 3 corresponds to the fourth channel. For more information on our naming convention, the reader is referred to Appendix A (page 121).

3.1 Constants

The following Constants are used throughout the driver code:

- `<S/UNI-4x622 ERROR CODES>`: error codes used throughout the driver code, returned by the API functions and used in the global error number field of the MDB and DDB. For a complete list of error codes, see Appendix B (page 125).
- `SUNI4x622_MAX_DEVS`: defines the maximum number of devices that can be supported by this driver. This constant must not be changed without a thorough analysis of the consequences to the driver code
- `SUNI4x622_MOD_START`, `SUNI4x622_MOD_IDLE`, `SUNI4x622_MOD_READY`: the three possible module states (stored in `stateModule`)
- `SUNI4x622_START`, `SUNI4x622_PRESENT`, `SUNI4x622_ACTIVE`, `SUNI4x622_INACTIVE`: the four possible device states (stored in `stateDevice`)

3.2 Structures Passed by the Application

These structures are defined for use by the application and are passed as argument to functions within the driver. These structures are the Module Initialization Vector (MIV), the Device Initialization Vector (DIV) and the ISR mask.

Module Initialization Vector: MIV

Passed via the `suni4x622ModuleOpen` call, this structure contains all the information needed by the driver to initialize and connect to the RTOS.

- `maxDevs` is used to inform the driver how many devices will be operating concurrently during this session. The number is used to calculate the amount of memory that will be allocated to the driver. The maximum value that can be passed is `SUNI4x622_MAX_DEVS` (see section 3.1).

Table 1: S/UNI-4x622 Module Initialization Vector: sSUNI4x622_MIV

Field Name	Field Type	Field Description
perrModule	INT4 *	(pointer to) errModule (see description in the MDB)
maxDevs	UINT2	Maximum number of devices supported during this session
maxInitProfs	UINT2	Maximum number of initialization profiles

Device Initialization Vector: DIV

Passed via the suni4x622Init call, this structure contains all the information needed by the driver to initialize a S/UNI-4x622 device. This structure is also passed via the suni4x622SetInitProfile call when used as an initialization profile.

- valid indicates that this initialization profile has been properly initialized and may be used by the USER. This field should be ignored when the DIV is passed directly.
- pollISR is a flag that indicates the type of interrupt servicing the driver is to use. The choices are 'polling' (SUNI4x622_POLL_MODE), and 'interrupt driven' (SUNI4x622_ISR_MODE). When configured in polling the interrupt capability of the device is NOT used, and the USER is responsible for calling suni4x622Poll periodically. The actual processing of the event information is the same for both modes.
- cbackSOH, cbackLOH, cbackRPOH, cbackPYLD, cbackFIFO, cbackIntfSys, cbackIntfLine and cbackAPS are used to pass the address of application functions that will be used by the DPR to inform the application code of pending events. If these fields are set as NULL, then any events that might cause the DPR to 'call back' the application will be processed during ISR processing but ignored by the DPR.

Table 2: S/UNI-4x622 Device Initialization Vector: sSUNI4x622_DIV

Field Name	Field Type	Field Description
valid	UINT2	Indicates that this structure is valid
pollISR	eSUNI4x622_ISR_MODE	Indicates the type of ISR / polling to do
cbackSOH	sSUNI4x622_CBACK	Address for the callback function for SOH events
cbackLOH	sSUNI4x622_CBACK	Address for the callback function for LOH events
cbackRPOH	sSUNI4x622_CBACK	Address for the callback function for RPOH events

Field Name	Field Type	Field Description
cbackPYLD	sSUNI4x622_CBACK	Address for the callback function for PYLD events
cbackFIFO	sSUNI4x622_CBACK	Address for the callback function for FIFO events
cbackIntfLine	sSUNI4x622_CBACK	Address for the callback function for Line Interface events
cbackIntfSys	sSUNI4x622_CBACK	Address for the callback function for System Interface events
cbackAPS	sSUNI4x622_CBACK	Address for the callback function for APS events
cfgGlobal	sSUNI4x622_CFG_GLOBAL	Global configuration block
cfgChan[4]	sSUNI4x622_CFG_CHAN	Channel configuration block (4 channels per device)

ISR Enable/Disable Mask

Passed via the suni4x622SetMask, suni4x622GetMask and suni4x622ClrMask calls, this structure contains all the information needed by the driver to enable and disable any of the interrupts in the S/UNI-4x622.

Table 3: S/UNI-4x622 ISR Mask: sSUNI4x622_MASK_ISR

Field Name	Field Type	Field Description
maskIntfSys	sSUNI4x622_MASK_ISR_INTF_SYS	Interrupt mask for System Interface
maskChan[4]	sSUNI4x622_MASK_ISR_CHAN	Interrupt mask for each channel (4 channels per device)
maskAPS[4]	sSUNI4x622_MASK_ISR_APS	Interrupt mask for each channel in the APS link (4 APS channels per device)

Table 4: S/UNI-4x622 ISR Mask: sSUNI4x622_MASK_ISR_CHAN

Field Name	Field Type	Field Description
maskSOH	sSUNI4x622_MASK_ISR_SOH	Interrupt mask for Section Overhead section
maskLOH	sSUNI4x622_MASK_ISR_LOH	Interrupt mask for Line Overhead section
maskRPOH	sSUNI4x622_MASK_ISR_RPOH	Interrupt mask for Receive Path Overhead section
maskPYLD	sSUNI4x622_MASK_ISR_PYLD	Interrupt mask for Payload Processor section
maskFIFO	sSUNI4x622_MASK_ISR_FIFO	Interrupt mask for FIFO Configuration section
maskIntfLine	sSUNI4x622_MASK_ISR_INTF_LINE	Interrupt mask for Line Interface section

Table 3: S/UNI-4x622 Section Overhead (SOH) ISR Mask: sSUNI4x622_MASK_ISR_SOH

Field Name	Field Type	Field Description
oof	UINT2	Out of frame
lof	UINT2	Loss of frame
los	UINT2	Loss of signal
sbipe	UINT2	Section BIP error
tiu	UINT2	Section trace unstable
tim	UINT2	Section trace mismatch

Table 4: S/UNI-4x622 Line Overhead (LOH) ISR Mask: sSUNI4x622_MASK_ISR_LOH

Field Name	Field Type	Field Description
lais	UINT2	Line alarm signal
lrdi	UINT2	Line remote defect

Field Name	Field Type	Field Description
psbf	UINT2	APS byte failure
coaps	UINT2	Change of APS bytes
coz1s1	UINT2	Change of synchronization status message
lbipe	UINT2	Line BIP error
lreie	UINT2	Line REI error
sdber	UINT2	Signal Defect
sfber	UINT2	Signal Failure

Table 5: S/UNI-4x622 Receive Path Overhead (RPOH) ISR Mask: sSUNI4x622_MASK__ISR_RPOH

Field Name	Field Type	Field Description
tiu	UINT2	Path trace unstable
tim	UINT2	Path trace mismatch
prpslmi	UINT2	Path signal label mismatch
prpslui	UINT2	Path signal label unstable
prdi	UINT2	Path remote defect indication
perdi	UINT2	Path enhanced remote defect indication
pbipe	UINT2	Path BIP-8 error
pfebe	UINT2	Path REI error
pais	UINT2	Path AIS state changes
ppse	UINT2	Positive Pointer Justification
pnse	UINT2	Negative Pointer Justification
ploptr	UINT2	Path Loss of pointer state changes
ardi	UINT2	AuxRDI state changes
uneq	UINT2	Trace identifier equipped state changes

Field Name	Field Type	Field Description
psl	UINT2	Path signal label changed
aisc	UINT2	Pointer AIS event
lopc	UINT2	Lost of pointer concatenation change
newptr	UINT2	New pointer received
illjreq	UINT2	Illegal Pointer Justification
discopa	UINT2	Discontinuous change of pointer
invndf	UINT2	Invalid NDF
illptr	UINT2	Illegal pointer
ndf	UINT2	NDF event

Table 6: S/UNI-4x622 ISR Mask: sSUNI4x622_MASK_ISR_PYLD

Field Name	Field Type	Field Description
lcd	UINT2	Change in loss of cell delineation
hcs	UINT2	Detection of corrected or uncorrected HCS error
fcs	UINT2	Detection of FCS error
rxcpxfer	UINT2	Transfer of received CP accumulated interval complete
txcpxfer	UINT2	Transfer of transmit CP accumulated counter data completed
abrt	UINT2	Reception of aborted packet
maxl	UINT2	Reception of packet exceeding maximum packet length
minl	UINT2	Reception of packet below minimum packet length
oocd	UINT2	Change in cell delineation state

Table 7: S/UNI-4x622 ISR Mask: sSUNI4x622_MASK_ISR_FIFO

Field Name	Field Type	Field Description
rxcpfovr	UINT2	Rx CP FIFO overrun

Field Name	Field Type	Field Description
rxfpfovr	UINT2	Rx FP FIFO overrun
txfpfudr	UINT2	Tx FP FIFO underrun

Table 10: S/UNI-4x622 ISR Mask: sSUNI4x622_MASK_ISR_INTF_LINE

Field Name	Field Type	Field Description
wansinten	UINT2	WANS phase detector averaging period has begun
lot	UINT2	Loss of transition
rool	UINT2	Recovered reference out of lock
dool	UINT2	Recovered data out of lock

Table11: S/UNI-4x622 ISR Mask: sSUNI4x622_MASK_ISR_INTF_SYS

Field Name	Field Type	Field Description
txop	UINT2	TSOP or TSEP is not asserted with the first or last word of a POS-PHY packet
unprov	UINT2	Detection of non-existent channel buffer during in-band addressing
cam	UINT2	Data field mismatch
tprty	UINT2	Tx Parity error
tsoc	UINT2	Start of cell re-alignment
fovr	UINT2	TUL3 FIFO overrun
funr	UINT2	RUL3 FIFO underrun

Table12: S/UNI-4x622 ISR Mask: sSUNI4x622_MASK_ISR_APS

Field Name	Field Type	Field Description
bip	UINT2	BIP-8 error
los	UINT2	Loss of signal

Field Name	Field Type	Field Description
lof	UINT2	Los of frame
oof	UINT2	Out of frame
lot	UINT2	Loss of transition
dool	UINT2	Recovered data out of lock
rool	UINT2	Recovered reference out of lock
ese	UINT2	Elastic store FIFO error
pj	UINT2	Pointer Justification

3.3 Structures in the Driver's Allocated Memory

These structures are defined and used by the driver and are part of the context memory allocated when the driver is opened. These structures are the Module Data Block (MDB), the Device Data Block (DDB).

Module Data Block: MDB

The MDB is the top-level structure for the module. It contains configuration data about the module level code and pointers to configuration data about the device level codes.

- errModule most of the module API functions return a specific error code directly. When the returned code is SUNI4x622_FAILURE, this indicates that the top-level function was not able to carry the specified error code back to the application. Under those circumstances, the proper error code is recorded in this element. The element is the first in the structure so that the USER can cast the MDB pointer into a INT4 pointer and retrieve the local error (this eliminates the need to include the MDB template into the application code).
- valid indicates that this structure has been properly initialized and may be read by the USER.
- stateModule contains the current state of the module and could be set to: SUNI4x622_MOD_START, SUNI4x622_MOD_IDLE or SUNI4x622_MOD_READY.

Table 8: S/UNI-4x622 Module Data Block: sSUNI4x622_MDB

Field Name	Field Type	Field Description
errModule	INT4	Global error Indicator for module calls
valid	UINT2	Indicates that this structure has been

Field Name	Field Type	Field Description
		initialized
stateModule	eSUNI4x622_MOD_STATE	Module state; can be one of the following IDLE or READY
maxDevs	UINT2	Maximum number of devices supported
numDevs	UINT2	Number of devices currently registered
maxInitProfs	UINT2	Maximum number of initialization profiles
pddb	sSUNI4x622_DDB *	(array of) Device Data Blocks (DDB) in context memory
pinitProfs	sSUNI4x622_DIV *	(array of) Initialization profiles in context memory

Device Data Block: DDB

The DDB is the top-level structure for each device. It contains configuration data about the device level code and pointers to configuration data about device level sub-blocks.

- errDevice most of the device API functions return a specific error code directly. When the returned code is SUNI4x622_FAILURE, this indicates that the top-level function was not able to carry the specific error code back top the application. In addition, some device functions do not return an error code. Under those circumstances, the proper error code is recorded in this element. The element is the first in the structure so that the USER can cast the DDB pointer to a INT4 pointer and retrieve the local error (this eliminates the need to include the DDB template in the application code).
- valid indicates that this structure has been properly initialized and may be read by the USER.
- stateDevice contains the current state of the device and could be set to: SUNI4x622_START, SUNI4x622_PRESENT, SUNI4x622_ACTIVE or SUNI4x622_INACTIVE.
- usrCtxt is a value that can be used by the USER to identify the device during the execution of the callback functions. It is passed to the driver when suni4x622Add is called and returned to the USER in the DPV when a callback function is invoked. The element is unused by the driver itself and may contain any value.

Table 9: S/UNI-4x622 Device Data Block: sSUNI4x622_DDB

Field Name	Field Type	Field Description
errDevice	INT4	Global error indicator for device calls
valid	UINT2	Indicates that this structure has been

Field Name	Field Type	Field Description
		initialized
stateDevice	eSUNI4x622_DEV_STATE	Device State; can be one of the following PRESENT, ACTIVE or INACTIVE
baseAddr	void *	Base address of the device
usrCtxt	sSUNI4x622_USR_CTXT	Stores the user's context for the device. It is passed as an input parameter when the driver invokes an application callback
profileNum	UINT2	Profile number used at initialization
pollISR	eSUNI4x622_ISR_MODE	Indicates the current type of ISR / polling
cbackSOH	sSUNI4x622_CBACK	Address for the callback function for SOH events
cbackLOH	sSUNI4x622_CBACK	Address for the callback function for LOH events
cbackRPOH	sSUNI4x622_CBACK	Address for the callback function for RPOH events
cbackPYLD	sSUNI4x622_CBACK	Address for the callback function for PYLD events
cbackFIFO	sSUNI4x622_CBACK	Address for the callback function for FIFO events
cbackIntfLine	sSUNI4x622_CBACK	Address for the callback function for Line Interface events
cbackIntfSys	sSUNI4x622_CBACK	Address for the callback function for System Interface events
cbackAPS	sSUNI4x622_CBACK	Address for the callback function for APS events
cfgGlobal	sSUNI4x622_CFG_GLOBAL	Global configuration block
cfgChan[4]	sSUNI4x622_CFG_CHAN	Channel configuration block (4 channels per device)
mask	sSUNI4x622_MASK_ISR	Interrupt Enable Mask

Device-wide Global Configuration

Table 10: S/UNI-4x622 Input/Output Configuration: sSUNI4x622_CFG_GLOBAL

Field Name	Field Type	Field Description
sonetsel	UINT1	Select SONET/SDH mode
cfgIntfSys	sSUNI4x622_CFG_INTF_SYS_GLOBAL	System Interface configuration block
cfgIntfLine	sSUNI4x622_CFG_INTF_LINE_GLOBAL	Line Interface configuration block

Per-Channel Configuration

Table 11: S/UNI-4x622 Channel Configuration: sSUNI4x622_CFG_CHAN

Field Name	Field Type	Field Description
cfgSOH	sSUNI4x622_CFG_SOH	Section Overhead Processor (SOH) configuration block
cfgLOH	sSUNI4x622_CFG_LOH	Line Overhead Processor (LOH) configuration block
cfgRPOH	sSUNI4x622_CFG_RPOH	Receive Path Overhead Processor (RPOH) configuration block
cfgTPOH	sSUNI4x622_CFG_TPOH	Transmit Path Overhead Processor (TPOH) configuration block
cfgPYLD	sSUNI4x622_CFG_PYLD	Payload Processor (PYLD) configuration block
cfgFIFO	sSUNI4x622_CFG_FIFO	FIFO configuration
cfgLine	sSUNI4x622_CFG_INTF_LINE	Line Interface configuration block

Per-Channel Section Overhead (SOH) Configuration

Table 12: S/UNI-4x622 Section Overhead Configuration: sSUNI4x622_CFG_SOH

Field Name	Field Type	Field Description
algo2	UINT1	Selects framing pattern used to determine and maintain the frame alignment
sblkbip	UINT1	Controls accumulation of section BIP errors
dds	UINT1	Rx descrambling
ds	UINT1	Tx scrambling
zeroen	UINT1	Selects whether all zero trace identifier messages accepted or ignored

Per-Channel Line Overhead (LOH) Configuration

Table 13: S/UNI-4x622 Line Overhead Configuration: sSUNI4x622_CFG_LOH

Field Name	Field Type	Field Description
laisdet	UINT1	Selects Line AIS detection algorithm
lrdidet	UINT1	Selects Line RDI detection algorithm
lbipword	UINT1	Selects accumulation of line BIP errors
sdlrdi	UINT1	Controls whether signal degrade can cause LRDI insertion
sflrdi	UINT1	Controls whether signal failure can cause LRDI insertion
loflrdi	UINT1	Controls whether loss of frame can cause LRDI insertion
loslrdi	UINT1	Controls whether loss of signal can cause LRDI insertion
rtimlrdi	UINT1	Controls whether section trace message mismatch can cause LRDI insertion
rtiulrdi	UINT1	Controls whether section trace message unstable can cause LRDI insertion
laislrdi	UINT1	Controls whether line AIS can cause LRDI insertion
autolfebe	UINT1	Controls whether line BIP errors can cause FEBE insertion

Field Name	Field Type	Field Description
allones	UINT1	Controls whether incoming AIS will force the downstream Sonet/SDH frame to all ones
sdins	UINT1	Controls whether SD can cause Line AIS insertion
sfins	UINT1	Controls whether SF can cause Line AIS insertion
lofins	UINT1	Controls whether LOF can cause Line AIS insertion
losins	UINT1	Controls whether LOS can cause Line AIS insertion
rtimins	UINT1	Controls whether section TIM can cause Line AIS insertion
rtiuins	UINT1	Controls whether section TIU can cause Line AIS insertion
dccais	UINT1	Controls whether LOS or LOF can force all ones in the DCC outputs

Per-Channel Receive Path Overhead (RPOH) Configuration

Table 14: S/UNI-4x622 Receive Path Overhead Configuration: sSUNI4x622_CFG_RPOH

Field Name	Field Type	Field Description
enss	UINT1	Selects SS bits are taking into account in the pointer interpreter state machine
sos	UINT1	Enables justification more than 3 frames ago
iinvcnt	UINT1	Selects behavior of the consecutive INV_POINT event counter
zeroen	UINT1	Selects whether all zero trace identifier messages accepted or ignored

Per-Channel Transmit Path Overhead (TPOH) Configuration

Table 15: S/UNI-4x622 Transmit Path Overhead Configuration: sSUNI4x622_CFG_TPOH

Field Name	Field Type	Field Description
persist	UINT1	Control of the persistence of the RDI asserted into the transmitted stream

Field Name	Field Type	Field Description
lcdprdi	UINT1	Controls whether loss of cell delineation can cause PRDI insertion
alrmprdi	UINT1	Controls whether LOS,LOF or LAIS can cause PRDI insertion
paisprdi	UINT1	Controls whether PAIS can cause PRDI insertion
pslmprdi	UINT1	Controls whether Path signal label mismatch can cause PRDI insertion
lopprdi	UINT1	Controls whether loss of pointer indications can cause PRDI insertion
lopconprdi	UINT1	Controls whether loss of pointer concatenation indications can cause PRDI insertion
ptiuprdi	UINT1	Controls whether path trace identifier unstable can cause PRDI insertion
ptimprdi	UINT1	Controls whether path trace identifier mismatch can cause PRDI insertion
paisconprdi	UINT1	Controls whether path AIS concatenation events can cause PRDI insertion
uneqprdi	UINT1	Controls whether unequipped path signal label can cause PRDI insertion
lcdeprdi	UINT1	Controls whether loss of cell delineation can cause EPRDI insertion
noalmeprdi	UINT1	Controls whether LOS,LOF or LAIS will disable EPRDI insertion
nopaiseprdi	UINT1	Controls whether PAIS will disable EPRDI insertion
pslmeprdi	UINT1	Controls whether Path signal label mismatch can cause EPRDI insertion
nolopeprdi	UINT1	Controls whether loss of pointer indications can disable EPRDI insertion
nolopconeprdi	UINT1	Controls whether loss of pointer concatenation indications will disable EPRDI insertion
ptiueprdi	UINT1	Controls whether path trace identifier unstable can cause EPRDI insertion

Field Name	Field Type	Field Description
ptimeprdi	UINT1	Controls whether path trace identifier mismatch can cause EPRDI insertion
paisconpais	UINT1	Controls whether AIS concatenation events can cause PAIS insertion
lopconpais	UINT1	Controls whether loss of pointer concatenation events can cause PAIS insertion
pslmpais	UINT1	Controls whether path signal label mismatch can cause PAIS insertion
pslupais	UINT1	Controls whether path signal label unstable can cause PAIS insertion
loppais	UINT1	Controls whether loss of signal can cause PAIS insertion
tiupais	UINT1	Controls whether path TIU can cause PAIS insertion
timpais	UINT1	Controls whether path TIM can cause PAIS insertion
autopfebe	UINT1	Controls whether path BIP errors can cause FEBE insertion

Per-Channel Payload Processor Configuration

Table 16: S/UNI-4x622 Payload Processor: sSUNI4x622_CFG_PYLD

Field Name	Field Type	Field Description
rxddscr	UINT1	RX descrambles payload
rxcpdiscor	UINT1	Disables ATM HCS error correction
rxcpidlepass	UINT1	RX ignore idle cell header pattern and mask for ATM cells
rxcpccdis	UINT1	Disables cell delineation and filtering
rxcplcdc	UINT2	RX LCD Count Threshold
rxcpidlehdr	UINT1	RX idle cell header
rxcpidlemask	UINT1	RX idle cell mask
rxfpfcssel	UINT1	RX FCS select
rxfpfcspass	UINT1	RX selects FCS stripping

Field Name	Field Type	Field Description
rxfpminpl	UINT1	RX minimum packet length
rxfpmaxpl	UINT2	RX maximum packet length
txdscr	UINT1	TX scrambles outgoing payload
txcpidlehdr	UINT1	TX idle cell header
txcpidlepyld	UINT1	TX idle cell payload
txfpfcssel	UINT1	TX FCS select
mode	UINT1	ATM or POS mode
txfpipgap	UINT1	The number of Flag Sequence characters inserted between each POS Frame

Per-Channel FIFO Configuration

Table 17: S/UNI-4x622 FIFO Configuration: sSUNI4x622_CFG_FIFO

Field Name	Field Type	Field Description
rxfpril	UINT1	RX FIFO overrun before frame receive
txfptil	UINT1	TX FIFO fill level before frame transmit

Per-Channel Clock Interface Configuration

Table 18: S/UNI-4x622 Clock Interface Configuration: sSUNI4x622_CFG_CLK

Field Name	Field Type	Field Description
loopt	UINT1	Selects REFCLK or recovered clock as tx source
dccsel	UINT1	Configures whether the DCC is configured for section or line DCC operation
tfpen	UINT1	Controls whether frame pulse input used for alignment

Per-Channel RALRM Configuration

Table 19: S/UNI-4x622 Clock Interface Configuration: sSUNI4x622_CFG_RALRM

Field Name	Field Type	Field Description
losen	UINT1	Controls whether LOS set RALRM output
lofen	UINT1	Controls whether LOF set RALRM output
oofen	UINT1	Controls whether OOF set RALRM output
laisen	UINT1	Controls whether line AIS set RALRM output
lrdien	UINT1	Controls whether Line RDIset RALRM output
sdberen	UINT1	Controls whether SD threshold event set RALRM output
sfberen	UINT1	Controls whether SF threshold event set RALRM output
stimen	UINT1	Controls whether section TIM set RALRM output
lopen	UINT1	Controls whether LOP set RALRM output
lcden	UINT1	Controls whether LCD set RALRM output
paisen	UINT1	Controls whether path AIS set RALRM output
prdien	UINT1	Controls whether path RDI set RALRM output
perdien	UINT1	Controls whether path ERDI set RALRM output
pslmen	UINT1	Controls whether path signal label mismatch set RALRM output
ptimen	UINT1	Controls whether path TIM set RALRM output
conen	UINT1	Controls whether pointer concatenation violation set RALRM output

Per-Channel Line Interface Configuration

Table 20: S/UNI-4x622 Line Interface Configuration: sSUNI4x622_CFG_INTF_LINE

Field Name	Field Type	Field Description
cfgClk	sSUNI4x622_CFG_CLK	Clock configuration block

Field Name	Field Type	Field Description
cfgRALRM	sSUNI4x622_CFG_RALRM	RALRM configuration block

Device-wide System Interface Configuration

Table 21: S/UNI-4x622 Global System Interface Configuration: sSUNI4x622_CFG_INTF_SYS_GLOBAL

Field Name	Field Type	Field Description
tul3prtyp	UINT1	TUL3 parity
tul3l3mode	UINT1	TUL3 mode
tul3atmsiglbl	UINT1	TUL3 ATM signal label
tul3possiglbl	UINT1	TUL3 POS signal label
rul3prtyp	UINT1	RUL3 parity
rul3l3mode	UINT1	RUL3 mode
rul3atmsiglbl	UINT1	RUL3 ATM signal label
rul3possiglbl	UINT1	RUL3 POS signal label
tul3atmfifodp	UINT1	TUL3 ATM FIFO depth
tul3posfifolwm	UINT1	TUL3 POS FIFO low water mark
tul3posfifohwm	UINT1	TUL3 POS FIFO high water mark
tul3cellform	UINT1	TUL3 cell size (52 or 56)
rul3cellform	UINT1	RUL3 cell size (52 or 56)
rul3pause	UINT1	RUL3 minimum time between POS transfer burst
rul3tran	UINT1	RUL3 maximum single-channel transfer

Device-Wide Line Interface Configuration

Table 22: S/UNI-4x622 Global Line Interface Configuration: sSUNI4x622_CFG_INTF_LINE_GLOBAL

Field Name	Field Type	Field Description
rsel	UINT1	Selects which channel is used as a clock source for the rx clock output pin
tsel	UINT1	Selects which channel is used as a clock source for the tx clock output pin

Table 23: S/UNI-4x622 Signal Failure Configuration: sSUNI4x622_CFG_SF

Field Name	Field Type	Field Description
sfcmode	UINT1	Clears alarm using a window size 8 times longer than the alarm declaration window size
sfsmode	UINT1	Saturates the BIP count on a per window basis
sfberten	UINT1	Automatic monitoring of line bit error rate threshold events by the SF BERM
sfsap	UINT4	SF Accumulation period
sfsth	UINT4	SF Saturation Threshold
sfdth	UINT4	SF Declaring Threshold
sfcth	UINT4	SF Clearing Threshold

Table 24: S/UNI-4x622 Signal Defect Configuration: sSUNI4x622_CFG_SD

Field Name	Field Type	Field Description
sdcmode	UINT1	Clears alarm using a window size 8 times longer than the alarm declaration window size
sdsmode	UINT1	Saturates the BIP count on a per window basis
sdberten	UINT1	Automatic monitoring of line bit error rate threshold events by the SF BERM

Field Name	Field Type	Field Description
sdsap	UINT4	SD Accumulation period
sdsth	UINT4	SD Saturation Threshold
sddth	UINT4	SD Declaring Threshold
sdcth	UINT4	SD Clearing Threshold

Table 25: S/UNI-4x622 Channel Status Block: sSUNI4x622_STATUS_CHAN

Field Name	Field Type	Field Description
statusSOH	sSUNI4x622_STATUS_SOH	Alarms, status and counts from the Section Overhead (SOH) section
statusLOH	sSUNI4x622_STATUS_LOH	Alarms, status and counts from the Line Overhead (LOH) section
statusRPOH	sSUNI4x622_STATUS_RPOH	Alarms, status and counts from the Receive Path Overhead (RPOH) Section
statusPYLD	sSUNI4x622_STATUS_PYLD	Alarms, status and counts from the Payload Section
statusLINE	sSUNI4x622_STATUS_INTF_LINE	Alarms, status and counts from Line Interface section

Section Overhead (SOH) Status

Table 26: S/UNI-4x622 Section Overhead Status: sSUNI4x622_STATUS_SOH

Field Name	Field Type	Field Description
oof	UINT1	Out of frame defect
lof	UINT1	Loss of frame defect
los	UINT1	Loss of signal defect
tiu	UINT1	Section trace identifier unstable
tim	UINT1	Section trace identifier mismatch

Line Overhead (LOH) Status

Table 27: S/UNI-4x622 Line Overhead Status: sSUNI4x622_STATUS_LOH

Field Name	Field Type	Field Description
k1	UINT1	Receive K1
k2	UINT1	Receive K2
s1	UINT1	Receive S1
laisdet	UINT1	Line alarm signal defect
lrdidet	UINT1	Line remote defect indication
sfber	UINT1	Signal Failure
sdber	UINT1	Signal Degrade
psbf	UINT1	APS byte failure

Receive Path Overhead (RPOH) Status

Table 28: S/UNI-4x622 Receive Path Overhead Processor Status: sSUNI4x622_STATUS_RPOH

Field Name	Field Type	Field Description
perdi	UINT1	Path enhanced RDI status(filtered received EPRDI bits,G1 bit 5,6 and 7)
rxptr	UINT2	Last payload pointer read from the receive stream
rxss	UINT1	Rx SS (DD) bits
txptr	UINT2	Current payload pointer being inserted in the tx stream
txss	UINT1	Tx SS bits
apsl	UINT1	Accepted path signal label
epsl	UINT1	Expected path signal label
plop	UINT1	Path lost of pointer status
pais	UINT1	Path AIS status

Field Name	Field Type	Field Description
prdi	UINT1	Path RDI status
tiu	UINT1	Path trace identifier unstable
tim	UINT1	Path trace identifier mismatch
uneq	UINT1	Equip status of the path signal label
pslu	UINT1	Path signal label unstable
pslm	UINT1	Path signal label mismatch
ardi	UINT1	Aux RDI status
aisc	UINT1	AIS concatenated
lopc	UINT1	Loss of pointer concatenated

Payload Processor Status

Table 30: S/UNI-4x622 Payload Status: sSUNI4x622_STATUS_PYLD

Field Name	Field Type	Field Description
rxcpovr	UINT1	RXCP accumulation transfer overrun
txcpovr	UINT1	TXCP accumulation transfer overrun
lcd	UINT1	Loss of cell delineation
oocd	UINT1	out of cell delineation

Clock Status

Table 29: S/UNI-4x622 Clock Status: sSUNI4x622_STATUS_CLK

Field Name	Field Type	Field Description
tclka	UINT1	TCLKA active
rclka	UINT1	RCLKA active
rfclka	UINT1	RFCLK active

Field Name	Field Type	Field Description
tfclka	UINT1	TFCLK active
refclka	UINT1	REFCLKA active

Line Interface Status

Table 30: S/UNI-4x622 Line Interface Status: sSUNI4x622_STATUS_INTF_LINE

Field Name	Field Type	Field Description
statusClk	sSUNI4x622_STATUS_CLK	Clock interface status

Counters (CNT)

Table 31: S/UNI-4x622 Counters: sSUNI4x622_CNTR_CHAN

Field Name	Field Type	Field Description
cntrSOH	sSUNI4x622_CNTR_SOH	Counters for Section Overhead (SOH) section
cntrLOH	sSUNI4x622_CNTR_LOH	Counters for Line Overhead (LOH) section
cntrRPOH	sSUNI4x622_CNTR_RPOH	Counters for Receive Path Overhead (RPOH) section
cntrPYLD	sSUNI4x622_CNTR_PYLD	Counters for Payload Processor (PYLD) section

Section Overhead (SOH) Counter

Table 32: S/UNI-4x622 Section Overhead (SOH) Counters: sSUNI4x622_CNTR_SOH

Field Name	Field Type	Field Description
sbe	UINT2	Section BIP error counter

Line Overhead (LOH) Counter

Table 33: S/UNI-4x622 Line Overhead (LOH) Counters: sSUNI4x622_CNTR_LOH

Field Name	Field Type	Field Description
lbe	UINT4	Line BIP error counter
lfe	UINT4	Line REI error counter

Receive Path Overhead (RPOH) Counter

Table 34: S/UNI-4x622 Receive Path Overhead (RPOH) Counters: sSUNI4x622_CNTR_RPOH

Field Name	Field Type	Field Description
pbe	UINT2	Path BIP error counter
pfe	UINT2	Path REI error counter

Payload Processor (PYLD) Counter

Table 35: S/UNI-4x622 Payload Processor Counters: sSUNI4x622_CNTR_PYLD

Field Name	Field Type	Field Description
rxcpchcs	UINT1	Rx corrected HCS error count
rxcpuhcs	UINT1	Rx uncorrected HCS error count
rxcprcell	UINT4	Rx cell count
rxcpicell	UINT4	Rx idle cell count
txcptcell	UINT4	Tx cell count
rxfprbyte	UINT4	Rx byte count
rxfprframe	UINT4	Rx frame count
rxfprabrf	UINT2	Rx aborted frame count
rxfprfcsef	UINT2	Rx FCS error frame count

Field Name	Field Type	Field Description
rxfprminlf	UINT2	Rx minimum length error frame count
rxfprmaxlf	UINT2	Rx maximum length error frame count
txfptbyte	UINT4	Tx byte count
txfptframe	UINT4	Tx frame count
txfptusrabf	UINT4	Tx user aborted frame count
txfptferabf	UINT4	Tx underrun/error aborted frame count

3.4 Structures Passed through RTOS Buffers

Interrupt-Service Vector: ISV

This buffer structure is used to capture the status of the device (during a poll or ISR processing) for use by the Deferred-Processing Routine (DPR). It is the template for all device registers that are involved in exception processing. It is the application's responsibility to create a pool of ISV buffers (using this template to determine the buffer's size) when the driver calls the USER-supplied `sysSuni4x622BufferStart` function. An individual ISV buffer is then obtained by the driver via `sysSuni4x622ISVBufferGet` and returned to the 'pool' via `sysSuni4x622ISVBufferRtn`.

Table 36: S/UNI-4x622 Interrupt-Service Vector: sSUNI4x622_ISV

Field Name	Field Type	Field Description
deviceHandle	sSUNI4x622_HNDL	Handle to the device in cause
mask	sSUNI4x622_MASK_ISR	ISR mask filled with interrupt status

Deferred-Processing Vector: DPV

This block is used in two ways. First it is used to determine the size of buffer required by the RTOS for use in the driver. Second it is the template for data that is assembled by the DPR and sent to the application code. Note: the application code is responsible for returning this buffer to the RTOS buffer pool.

The DPR reports events to the application using user-defined callbacks. The DPR uses each callback to report a functionally-related group of events. Refer to page 104 for a description of the S/UNI-4x622 callback functions, and Appendix C (page 126) for a list of events.

Table 37: S/UNI-4x622 Deferred-Processing Vector: sSUNI4x622_DPV

Field Name	Field Type	Field Description
event	SUNI4x622_DPR_EVENT	Event being reported
cause	UINT2	Reason for the Event

3.5 Global Variable

Although most of the variables within the driver are not meant to be used by the application code, there is one global variable that can be of great use to the application code.

suni4x622Mdb: A global pointer to the Module Data Block (MDB). The content of this global variable should be considered read-only by the application.

- errModule: This structure element is used to store an error code that specifies the reason for an API function's failure. The field is only valid for functions that do not return an error code or when a value of SUNI4x622_FAILURE is returned.
- stateModule: This structure element is used to store the module state (as shown in Figure 3).
- pddb[ ]: An array of pointers to the individual Device Data Blocks. The USER is cautioned that a DDB is only valid if the valid flag is set. Note that the array of DDBs is in no particular order.
    - errDevice: This structure element is used to store an error code that specifies the reason for an API function's failure. The field is only valid for functions that do not return an error code or when a value of SUNI4x622_FAILURE is returned.
    - stateDevice: This structure element is used to store the device state (as shown in Figure 3).

4 APPLICATION PROGRAMMING INTERFACE

This section provides a detailed description of each function that is a member of the S/UNI-4x622 driver Application Programming Interface (API).

The API functions typically execute in the context of an application task.

Note: These functions are not re-entrant. This means that two application tasks can not invoke the same API at the same time. However the driver protects its data structures from concurrent accesses by the application and the DPR task.

4.1 Module Management

The module management is a set of API functions that are used by the application to open, start, stop and close the driver module. These functions will take care of initializing the driver, allocating memory and all RTOS resources needed by the driver. They are also used to change the module state. For more information on the module states see the state diagram on page 19. For a typical module management flow diagram see page 21.

Opening the Driver Module: suni4x622ModuleOpen

This function performs module level initialization of the device driver. This involves allocating all of the memory needed by the driver and initializing the internal structures.

Prototype `INT4 suni4x622ModuleOpen(sSUNI4x622_MIV *pmiv)`

Inputs `pmiv` : (pointer to) Module Initialization Vector

Outputs Places the address of the MDB into the MIV passed by the application

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_MODULE_STATE`
`SUNI4x622_ERR_INVALID_MIV`
`SUNI4x622_ERR_MEM_ALLOC`

Valid States `SUNI4x622_MOD_START`

Side Effects Changes the MODULE state to `SUNI4x622_MOD_IDLE`

Closing the Driver Module: suni4x622ModuleClose

This function performs module level shutdown of the driver. This involves deleting all devices being controlled by the driver (by calling `suni4x622Delete` for each device) and de-allocating all the memory allocated by the driver.

Prototype `INT4 suni4x622ModuleClose(void)`

Inputs None

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_MODULE_STATE`

Valid States `SUNI4x622_MOD_IDLE`, `SUNI4x622_MOD_READY`

Side Effects Changes the MODULE state to `SUNI4x622_MOD_START`

Starting the Driver Module: suni4x622ModuleStart

This function connects the RTOS resources to the driver. This involves allocating semaphores and timers, initializing buffers and installing the ISR handler and DPR task. Upon successful return from this function the driver is ready to add devices.

Prototype `INT4 suni4x622ModuleStart(void)`

Inputs None

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_MODULE_STATE`
`SUNI4x622_ERR_MEM_ALLOC`
`SUNI4x622_ERR_INT_INSTALL`

Valid States `SUNI4x622_MOD_IDLE`

Side Effects Changes the MODULE state to `SUNI4x622_MOD_READY`

Stopping the Driver Module: suni4x622ModuleStop

This function disconnects the RTOS resources from the driver. This involves de-allocating semaphores and timers, freeing-up buffers and uninstalling the ISR handler and the DPR task. If there are any registered devices, `suni4x622Delete` is called for each.

Prototype `INT4 suni4x622ModuleStop(void)`

Inputs None

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_MODULE_STATE`

Valid States `SUNI4x622_MOD_READY`

Side Effects Changes the MODULE state to `SUNI4x622_MOD_IDLE`

4.2 Profile Management

This section describes the functions that add, get and clear an initialization profile. Initialization profiles allow the user to store pre-defined Device Initialization Vectors (DIV) that are validated ahead of time. When the device initialization function is invoked only a profile number needs to be passed. This method simplifies and expedites the initialization process.

Adding an Initialization Profile: suni4x622AddInitProfile

This function creates an initialization profile that is stored by the driver. A device can now be initialized by simply passing the initialization profile number.

Prototype
```
INT4 suni4x622AddInitProfile(sSUNI4x622_DIV
*pProfile, UINT2 *pProfileNum)
```

Inputs
`pProfile` : (pointer to) initialization profile being added
`pProfileNum` : (pointer to) profile number to be assigned by the driver

Outputs
`pProfileNum` : (pointer to) profile number assigned by the driver

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_MODULE_STATE`
`SUNI4x622_ERR_INVALID_ARG`
`SUNI4x622_ERR_INVALID_PROFILE`
`SUNI4x622_ERR_PROFILES_FULL`

Valid States `SUNI4x622_MOD_IDLE, SUNI4x622_MOD_READY`

Side Effects None

Getting an Initialization Profile: suni4x622GetInitProfile

This function gets the content of an initialization profile given its profile number.

Prototype
```
INT4 suni4x622GetInitProfile(UINT2 profileNum,
sSUNI4x622_DIV *pProfile)
```

Inputs
`profileNum` : initialization profile number
`pProfile` : (pointer to) initialization profile

Outputs
`pProfile` : (pointer to) contents of the corresponding profile

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_MODULE_STATE`
`SUNI4x622_ERR_INVALID_ARG`
`SUNI4x622_ERR_INVALID_PROFILE_NUM`

Valid States SUNI4x622_MOD_IDLE, SUNI4x622_MOD_READY

Side Effects None

Deleting an Initialization Profile: suni4x622DeleteInitProfile

This function deletes an initialization profile given its profile number.

Prototype INT4 suni4x622DeleteInitProfile(UINT2 profileNum)

Inputs profileNum : initialization profile number

Outputs None

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_MODULE_STATE
SUNI4x622_ERR_INVALID_PROFILE_NUM

Valid States SUNI4x622_MOD_IDLE, SUNI4x622_MOD_READY

Side Effects None

4.3 Device Management

The device management is a set of API functions that are used by the application to control the device. These functions take care of initializing a device in a specific configuration, enabling the device general activity as well as enabling interrupt processing for that device. They are also used to change the software state for that device. For more information on the device states see the state diagram on page 19. For a typical device management flow diagram see page 22.

Adding a Device: suni4x622Add

This function verifies the presence of a new device in the hardware then returns a handle back to the user. The device handle is passed as a parameter of most of the device API Functions. It's used by the driver to identify the device on which the operation is to be performed.

Prototype sSUNI4x622_HNDL suni4x622Add(sSUNI4x622_USR_CTXT usrCtxt, void *pBaseAddr, INT4 **perrDevice)

Inputs usrCtxt : user context for this device
pBaseAddr : (pointer to) base address of the device
perrDevice : (pointer to) an area of memory

Outputs ERROR code written to the MDB on failure
SUNI4x622_ERR_INVALID_MODULE_STATE
SUNI4x622_ERR_INVALID_ARG
SUNI4x622_ERR_DEVS_FULL
SUNI4x622_ERR_DEV_ALREADY_ADDED
SUNI4x622_ERR_INVALID_DEV

`perrDevice` : (pointer to) `errDevice` (inside the DDB)

Returns Success = Device Handle (to be used bas an argument to most of the S/UNI-4x622 APIs)
Failure = NULL (pointer)

Valid States `SUNI4x622_MOD_READY`

Side Effects Changes the DEVICE state to `SUNI4x622_PRESENT`

Deleting a Device: suni4x622Delete

This function removes the specified device from the list of devices being controlled by the S/UNI-4x622 driver. Deleting a device involves invalidating the DDB for that device and releasing its associated device handle.

Prototype
```
INT4 suni4x622Delete(sSUNI4x622_HNDL deviceHandle)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEV`

Valid States `SUNI4x622_PRESENT, SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects Changes the DEVICE state to `SUNI4x622_PRESENT`

Initializing a Device: suni4x622Init

This function initializes the Device Data Block (DDB) associated with that device during `suni4x622Add`, applies a soft reset to the device and configures it according to the DIV passed by the application. If the DIV is passed as a NULL the profile number is used. A profile number of zero indicates that all the register bits are to be left in their default state (after a soft reset). Note that the profile number is ignored UNLESS the passed DIV is NULL.

Prototype
```
INT4 suni4x622Init(sSUNI4x622_HNDL deviceHandle,
sSUNI4x622_DIV *pdiv, UINT2 profileNum)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`pdiv` : (pointer to) Device Initialization Vector
`profileNum` : profile number (ignored if `pdiv` is NULL)

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_PROFILE_NUM`

SUNI4x622_ERR_INVALID_DIV

Valid States SUNI4x622_PRESENT

Side Effects Changes the DEVICE state to SUNI4x622_INACTIVE

Updating the Configuration of a Device: suni4x622Update

This function updates the configuration of the device as well as the Device Data Block (DDB) associated with that device according to the DIV passed by the application. The only difference between suni4x622Update and suni4x622Init is that no soft reset will be applied to the device.

Prototype
```
INT4 suni4x622Update(sSUNI4x622_HNDL deviceHandle,
sSUNI4x622_DIV *pdiv, UINT2 profileNum)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
pdiv	: (pointer to) Device Initialization Vector
profileNum	: profile number (ignored if pdiv is NULL)

Outputs None

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_PROFILE_NUM
SUNI4x622_ERR_INVALID_DIV

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Resetting a Device: suni4x622Reset

This function applies a software reset to the S/UNI-4x622 device. Also resets all the DDB contents (except for the user context). This function is typically called before re-initializing the device (via suni4x622Init).

Prototype
```
INT4 suni4x622Reset(sSUNI4x622_HNDL deviceHandle)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)

Outputs None

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV

Valid States SUNI4x622_PRESENT, SUNI4x622_ACTIVE,
SUNI4x622_INACTIVE

Side Effects Changes the DEVICE state to `SUNI4x622_PRESENT`

Activating a Device: suni4x622Activate

This function restores the state of a device after a de-activate. Interrupts may be re-enabled.

Prototype `INT4 suni4x622Activate(sSUNI4x622_HNDL deviceHandle)`

Inputs `deviceHandle` : device handle (from `suni4x622Add`)

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`

Valid States `SUNI4x622_INACTIVE`

Side Effects Changes the DEVICE state to `SUNI4x622_ACTIVE`

De-Activating a Device: suni4x622DeActivate

This function de-activates the device from operation. Interrupts are masked and the device is put into a quiet state via enable bits.

Prototype `INT4 suni4x622DeActivate(sSUNI4x622_HNDL deviceHandle)`

Inputs `deviceHandle` : device handle (from `suni4x622Add`)

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`

Valid States `SUNI4x622_ACTIVE`

Side Effects Changes the DEVICE state to `SUNI4x622_INACTIVE`

4.4 Device Read and Write

Reading from Device Registers: suni4x622Read

This function reads a register of a specific S/UNI-4x622 device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then reads the contents of this address location using the system specific macro, `sysSuni4x622Read`. Note that a failure to read returns a zero and any error indication is written to the associated DDB.

Prototype
```
UINT1 suni4x622Read(sSUNI4x622_HNDL deviceHandle,
UINT2 regNum)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`regNum` : register number

Outputs
ERROR code written to the MDB
`SUNI4x622_ERR_INVALID_DEV`
ERROR code written to the DDB
`SUNI4x622_ERR_INVALID_REG`

Returns
Success = value read
Failure = 0

Valid States
`SUNI4x622_PRESENT, SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects
May affect registers that change after a read operation

Writing to Device Registers: suni4x622Write

This function writes to a register of a specific S/UNI-4x622 device by providing the register number. This function derives the actual address location based on the device handle and register number inputs. It then writes the contents of this address location using the system specific macro, `sysSuni4x622Write`. Note that a failure to write returns a zero and any error indication is written to the DDB.

Prototype
```
UINT1 suni4x622Write(sSUNI4x622_HNDL deviceHandle,
UINT2 regNum, UINT1 value)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`regNum` : register number
`value` : value to be written

Outputs
ERROR code written to the MDB
`SUNI4x622_ERR_INVALID_DEV`
ERROR code written to the DDB
`SUNI4x622_ERR_INVALID_REG`

Returns
Success = value written

Failure = 0

Valid States `SUNI4x622_PRESENT, SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the configuration of the device

Reading from a block of Device Registers: suni4x622ReadBlock

This function reads a register block of a specific S/UNI-4x622 device by providing the starting register number, and the size to read. This function derives the actual start address location based on the device handle and starting register number inputs. It then reads the contents of this data block using multiple calls to the system specific macro, `sysSuni4x622Read`. Note that any error indication is written to the DDB. It is the USER's responsibility to allocate enough memory for the block read.

Prototype
```
void suni4x622ReadBlock(sSUNI4x622_HNDL deviceHandle,
UINT2 startRegNum, UINT2 size, UINT1 *pblock)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`startRegNum`	: starting register number
`size`	: size of the block to read
`pblock`	: (pointer to) the block to read

Outputs

ERROR code written to the DDB
`SUNI4x622_ERR_INVALID_DEV`
ERROR code written to the DDB
`SUNI4x622_ERR_INVALID_ARG`
`SUNI4x622_ERR_INVALID_REG`

`pblock`	: (pointer to) the block read

Returns None

Valid States `SUNI4x622_PRESENT, SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May affect registers that change after a read operation

Writing to a Block of Device Registers: suni4x622WriteBlock

This function writes to a register block of a specific S/UNI-4x622 device by providing the starting register number and the block size. This function derives the actual starting address location based on the device handle and starting register number inputs. It then writes the contents of this data block using multiple calls to the system specific macro, `sysSuni4x622Write`. A bit from the passed block is only modified in the device's registers if the corresponding bit is set in the passed mask. Note that any error indication is written to the DDB

Prototype
```
void suni4x622WriteBlock(sSUNI4x622_HNDL
deviceHandle, UINT2 startRegNum, UINT2 size, UINT1
*pblock, UINT1 *pmask)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`startRegNum` : starting register number
`size` : size of block to read
`pblock` : (pointer to) block to write
`pmask` : (pointer to) mask

Outputs ERROR code written to the DDB
`SUNI4x622_ERR_INVALID_DEV`
ERROR code written to the DDB
`SUNI4x622_ERR_INVALID_ARG`
`SUNI4x622_ERR_INVALID_REG`

Returns None

Valid States `SUNI4x622_PRESENT, SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the configuration of the device

4.5 Section Overhead (SOH)

The Section Overhead section provides functions to control and monitor the section overhead processing. Read / Write access is given to the section trace message (J0). This message is compared with a configurable reference and mismatches are reported. For diagnostic purposes, errors can be introduced in the section overhead bytes.

Writing the J0 Byte: suni4x622SOHWriteJ0

This function writes the J0 byte into the transmit section overhead.

Prototype `INT4 suni4x622SOHWriteJ0(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT1 J0)`

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`J0` : J0 byte to write

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Reading and Setting the Section Trace Message : suni4x622SOHTraceMsg

This function retrieves and sets the section trace message (J0) in the Sonet/SDH Section Trace Buffer.

Note: It is the USER's responsibility to ensure that the message pointer points to an area of memory large enough to hold the returned data.

Prototype `INT4 suni4x622SOHTraceMsg(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 acctyp, UINT1* pJ0)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`acctyp`	: type of access 0 = write tx section trace msg 1 = read rx accepted section trace msg 2 = read rx captured section trace msg 3 = write rx expected section trace msg
`pJ0`	: (pointer to) the section trace message

Outputs

`pJ0`	: (pointer to) section trace message

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`
`SUNI4x622_ERR_POLL_TIMEOUT`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Forcing A1 Error : suni4x622SOHForceA1

When the enable flag is set, this function introduces framing errors in the A1 bytes. When the enable flag is not set, this function resumes normal processing.

Prototype `INT4 suni4x622SOHForceA1(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`enable`	: flag to start/stop A1 error insertion

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`

SUNI4x622_ERR_INVALID_CHAN

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Forcing B1 Error: suni4x622SOHForceB1

This function inserts the B1 BIP-8 errors byte to be inserted to the section overhead.

Prototype
```
INT4 suni4x622SOHForceB1(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT2 enable)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number
enable	: flag to start/stop B1 error insertion

Outputs None

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Forcing OOF: suni4x622SOHForceOOF

This function forces the Section Overhead Processor temporarily out of frame. The Section Overhead Processor will attempt to lock back onto the frame.

Prototype
```
INT4 suni4x622SOHForceOOF(sSUNI4x622_HNDL
deviceHandle, UINT1 channel)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number

Outputs None

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Forcing LOS: suni4x622SOHForceLOS

When the enable flag is set, this function forces a loss of signal condition in the data stream. When the enable flag is not set, this function resumes normal processing.

Prototype `INT4 suni4x622SOHForceLOS(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`enable`	: flag to start/stop a loss of signal condition

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

4.6 Line Overhead (LOH)

The Line Overhead section provides functions to configure and monitor the line overhead on both the receive and transmit sides. Read / Write access is given to the APS bytes (K1 and K2) and most other overhead bytes. Signal failure and signal degrade can be monitored. For diagnostic purposes, errors can be introduced in the line overhead bytes. Additional functions are provided to automatically insert line RDI and line AIS.

Configuring SF Error Monitor: suni4x622LOHSFCfg

This function configures the Signal Failure BERM automatic monitoring of line bit error rate threshold events.

Prototype `INT4 suni4x622LOHSFCfg(sSUNI4x622_HNDL deviceHandle, UINT1 channel, sSUNI4x622_CFG_SF *psfcfg)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`psfcfg`	: (pointer to) SF configuration block

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Configuring SD Error Monitor: suni4x622LOHSDCfg

This function configures the Signal Defect BERM automatic monitoring of line bit error rate threshold events.

Prototype
```
INT4 suni4x622LOHSDCfg(sSUNI4x622_HNDL deviceHandle,
UINT1 channel, sSUNI4x622_CFG_SD *psdcfg)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number
psdcfg	: (pointer to) SD configuration block

Outputs None

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Writing the K1K2 Byte: suni4x622LOHWriteK1K2

This function writes the K1, K2 byte into the line overhead.

Prototype
```
INT4 suni4x622LOHWriteK1K2(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT1 K1, UINT1 K2)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number
K1	: K1 byte to write
K2	: K2 byte to write

Outputs None

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Reading the K1K2 Byte: suni4x622LOHReadK1K2

This function reads the K1K2 byte from the line overhead.

Prototype
```
INT4 suni4x622LOHReadK1K2(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT1 *pK1, UINT1 *pK2)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pK1`	: (pointer to) store K1 byte read
`pK2`	: (pointer to) store K2 byte read

Outputs

`pK1`	: (pointer to) K1 byte read
`pK2`	: (pointer to) K2 byte read

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Writing the S1 Byte: suni4x622LOHWriteS1

This function writes the S1 byte into the line overhead.

Prototype
```
INT4 suni4x622LOHWriteS1(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT1 S1)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`S1`	: S1 byte to write

Outputs None

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Reading the S1 Byte: suni4x622LOHReadS1

This function reads the S1 byte from the line overhead.

Prototype `INT4 suni4x622LOHReadS1(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT1 *pS1)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pS1`	: (pointer to) store S1 byte read

Outputs

`pS1`	: (pointer to) S1 byte read

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Forcing Line AIS: suni4x622LOHForceAIS

This function forces line AIS in the transmit direction.

Prototype `INT4 suni4x622LOHForceAIS(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`enable`	: flag to start/stop line AIS insertion

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Forcing B2 Error: suni4x622LOHForceB2

This function forces B2 BIP-8 errors into the line overhead.

Prototype `INT4 suni4x622LOHForceB2(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`enable` : flag to start/stop B2 error insertion

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Forcing Line RDI: suni4x622LOHForceRDI

This function forces the line RDI into the line overhead.

Prototype
```
INT4 suni4x622LOHForceRDI(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT2 enable)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`enable` : flag to start/stop line RDI insertion

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

4.7 Path Overhead (RPOH, TPOH)

The Path Overhead Processor is responsible for pointer interpretation, path overhead processing, synchronous payload envelope, path level alarm and performance monitoring on both receive and transmit sides. The Path Overhead section configures and monitors the path overhead on both receive and transmit sides. Write access is given to the path trace message (J1) and the path signal label (C2) and other overhead bytes. For diagnostic purposes, errors can be introduced in the path overhead bytes. Additional functions are provided to automatically insert path AIS, force generation of outgoing justification events.

Retrieving and Setting the Path Trace Messages: suni4x622POHTraceMsg

This function retrieves and sets the current path trace message in the Sonet/SDH Path Trace Buffer. Note: It is the USER's responsibility to make sure that the message pointer points to an area of memory large enough to hold the returned data.

Prototype
```
INT4 suni4x622POHTraceMsg(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT2 acctyp, UINT1*
pJ1)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`acctyp`	: type of access 0 = write tx path trace msg 1 = read rx accepted path trace msg 2 = read rx captured path trace msg 3 = write rx expected path trace msg
`pJ1`	: (pointer to) the path trace message

Outputs

`pJ1`	: (pointer to) updated path trace message

Returns

Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`
`SUNI4x622_ERR_POLL_TIMEOUT`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Writing the J1 Byte: suni4x622TPOHWriteJ1

This function writes the J1 byte into the path overhead.

Prototype
```
INT4 suni4x622TPOHWriteJ1(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT1 J1)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`J1`	: J1 byte to write

Outputs None

Returns

Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Writing the C2 Byte: suni4x622TPOHWriteC2

This function writes the C2 byte into the path overhead.

Prototype
```
INT4 suni4x622TPOHWriteC2(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT1 C2)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number
C2	: C2 byte to write

Outputs None

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Writing the New Data Flag Bits: suni4x622TPOHWriteNDF

This function writes the passed new data flag bits (NDF[3:0]) in the NDF bit positions.

Prototype
```
INT4 suni4x622TPOHWriteNDF(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT2 enable, UINT1 ndf)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number
enable	:flag to start/stop inserting the NDF value passed in this function
ndf	: NDF value (lower nibble)

Outputs None

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Writing SS Bits: suni4x622TPOHWriteSS

This function writes the passed SS bits (SS[1:0]) in the SS bit positions.

Prototype `INT4 suni4x622TPOHWriteSS(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT1 ss)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`ss`	: SS bits value(bit 0 and bit 1)

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Inserting a Pointer Value: suni4x622TPOHInsertTxPtr

This function enables the insertion of the pointer value passed in argument into the H1 and H2 bytes of the transmit stream. As a result, the upstream payload mapping circuitry and a valid SPE can continue functioning and generating normally.

Prototype `INT4 suni4x622TPOHInsertTxPtr(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 aptr)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`aptr`	: pointer value to insert in (H1, H2)

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Force Path BIP-8 Errors: suni4x622TPOHForceB3

This function forces the B3 error in the transmit stream.

Prototype `INT4 suni4x622THPPForceB3(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`enable`	: flag to start/stop B3 masking

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Forcing Pointer Justification: suni4x622TPOHForcePJ

The function forces single positive or negative pointer justification.

Prototype `INT4 suni4x622TPOHForcePJ(sSUNI4x622_HNDL deviceHandle, UINT1 channel, INT1 type)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`type`	: type of Pointer Justification event: –1 = negative Pointer Justification +1 = positive Pointer Justification

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Forcing Path RDI: suni4x622TPOHForceRDI

This function enables the insertion of the Path Alarm Indication Signal (PRDI) in the transmit stream.

Prototype `INT4 suni4x622TPOHForceRDI(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`enable` : flag to start/stop path RDI insertion

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Forcing Path ERDI: suni4x622TPOHForceERDI

This function enables the insertion of the Enhanced Path Alarm Indication Signal (EPRDI) in the transmit stream.

Prototype
```
INT4 suni4x622TPOHForceERDI(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT2 enable)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`enable` : flag to start/stop path ERDI insertion

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Forcing Path ARDI: suni4x622TPOHForceARDI

This function enables the insertion of the Auxiliary Path Alarm Indication Signal (APRDI) in the transmit stream.

Prototype
```
INT4 suni4x622TPOHForceARDI(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT2 enable)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`enable` : flag to start/stop path ARDI insertion

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Forcing Path AIS: suni4x622TPOHForceAIS

This function enables the insertion of the Path Alarm Indication Signal (PAIS) in the transmit stream. The synchronous payload envelope and the pointer bytes (H1 – H3) are set to all ones.

Prototype
```
INT4 suni4x622TPOHForceAIS(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, UINT2 enable)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`enable`	: flag to start/stop path AIS insertion

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

4.8 Payload Processor

The Payload Processor performs both ATM and PPP processing.

Setting Payload configuration parameters: suni4x622PyldCfg

This function sets up the minimum and maximum packet length, cell header and mask for packet/cell payload configuration.

Prototype
```
INT4 suni4x622PyldCfg(sSUNI4x622_HNDL deviceHandle,
UINT1 channel, sSUNI4x622_CFG_PYLD *ppyldcfg)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number

`ppyldcfg` : (pointer to) payload configuration parameters

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

4.9 Interface Configuration

The Interface Configuration provides FIFO management to separate the line side timing from the high layer ATM/POS link layer timing, the line and system interface configuration.

Resetting the Receive/Transmit FIFO: suni4x622FIFOReset

This function resets the receive and/or transmit FIFO.

Prototype
```
INT4 suni4x622FIFOReset(sSUNI4x622_HNDL deviceHandle,
UINT1 channel, UINT1 fifotype)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`fifotype` : =0 if RX; = 1 if TX; = 2 if both

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Configuring the Receive and Transmit FIFO: suni4x622FIFOCfg

This function configures the FIFO based on the FIFO configuration supplied.

Prototype
```
INT4 suni4x622FIFOCfg(sSUNI4x622_HNDL deviceHandle,
UINT1 channel, sSUNI4x622_CFG_FIFO *pfifocfg)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`pfifocfg` : (pointer to) FIFO configuration block

Outputs None

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Configuring the System interface: suni4x622SysIntfCfg

This function configures the system interface based on the configuration parameters.

Prototype
```
INT4 suni4x622SysIntfCfg(sSUNI4x622_HNDL
deviceHandle, sSUNI4x622_CFG_INTF_SYS_GLOBAL *psyscfg
)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`psyscfg` : (pointer to) system interface configuration block

Outputs None

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Configuring the Device-Wide Line interface: suni4x622IntfLineCfg

This function configures the device-wide line interface based on the configuration parameters.

Prototype
```
INT4 suni4x622IntfLineCfg(sSUNI4x622_HNDL
deviceHandle, sSUNI4x622_CFG_INTF_LINE_GLOBAL
*plinecfg )
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`plinecfg` : (pointer to) device-wide line interface configuration block

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Resetting the TFCLK DLL: suni4x622IntfSysResetTDLL

This function resets the TFCLK DLL.

Prototype
```
INT4 suni4x622IntfSysResetTDLL(sSUNI4x622_HNDL
deviceHandle)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Resetting the RFCLK DLL: suni4x622IntfSysResetRDLL

This function resets the RFCLK DLL.

Prototype
```
INT4 suni4x622IntfSysResetRDLL(sSUNI4x622_HNDL
deviceHandle)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

4.10 Automatic Protection Configuration

The Automatic Protection Configuration section is responsible for configuring the S/UNI-4x622 to use the APS ports.

Configuring APS Working/Protect Mate: suni4x622APSCfg

When enable is set, this function enables the S/UNI-4x622 to operate as an APS working/protect mate under a failed condition. When enable is not set, this function resumes normal operation.

Prototype `INT4 suni4x622APSCfg(sSUNI4x622_HNDL deviceHandle, UINT1 mode, UINT1 enable)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`mode`	: =0 if working mate; =1 if protect mate
`enable`	: start/stop APS operation

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Configuring the Source Channel for the Given Channel Receive Path: suni4x622RPCfg

The function is used to control the source channel for the receive path.

Prototype `INT4 suni4x622RPCfg(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT1 srcChan)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: receive path channel number
`srcChan`	: source channel number

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Configuring the Source Channel for the Given Channel Transmit Path: suni4x622TPCfg

The function is used to control the source channel for the transmit path.

Prototype `INT4 suni4x622TPCfg(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT1 srcChan)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: transmit path channel number
`srcChan`	: source channel number

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Enable or disable the channel APS cross connect: suni4x622APSXcnntCfg

The function is used to control the channel APS cross connect.

Prototype `INT4 suni4x622APSXcnntCfg(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT1 enable)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`enable`	: enable or disable

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Resetting APS Receive Link: suni4x622APSResetRxLink

The function is used to reset the receive APS link.

Prototype `INT4 suni4x622APSResetRxLink(sSUNI4x622_HNDL`

```
deviceHandle, UINT1 link)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`link` : APS link number

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_APS_LINK`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Resetting APS Transmit Link: suni4x622APSResetTxLink

The function is used to reset the transmit APS link.

Prototype
```
INT4 suni4x622APSResetTxLink(sSUNI4x622_HNDL
deviceHandle, UINT1 link)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`link` : APS link number

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_APS_LINK`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

4.11 Interrupt Service Functions

This section describes interrupt-service functions that perform the following tasks:

- Set, get and clear the interrupt enable mask
- Read and process the interrupt-status registers
- Poll and process the interrupt-status registers

See page 23 for an explanation of our interrupt servicing architecture.

Configuring ISR Processing: suni4x622ISRConfig

Allows the USER to configure how ISR processing is to be handled: polling (SUNI4x622_POLL_MODE) or interrupt driven (SUNI4x622_ISR_MODE). If polling is selected, the USER is responsible for calling periodically suni4x622Poll to collect exception data from the device.

Prototype INT4 suni4x622ISRConfig(sSUNI4x622_HNDL deviceHandle, eSUNI4x622_ISR_MODE mode)

Inputs
deviceHandle : device handle (from suni4x622Add)
mode : mode of operation

Outputs None

Returns
Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Getting Device Interrupt Enable Mask: suni4x622GetMask

This function returns the contents of the interrupt mask registers of the S/UNI-4x622 device.

Prototype INT4 suni4x622GetMask(sSUNI4x622_HNDL deviceHandle, sSUNI4x622_MASK_ISR *pmask)

Inputs
deviceHandle : device handle (from suni4x622Add)
pmask : (pointer to) mask structure

Outputs
pmask : (pointer to) updated mask structure

Returns
Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Setting Device Interrupt Enable Mask: suni4x622SetMask

This function sets the contents of the interrupt mask registers of the S/UNI-4x622 device. Any bits that are set in the passed structure are set in the associated S/UNI-4x622 registers.

Prototype `INT4 suni4x622SetMask(sSUNI4x622_HNDL deviceHandle, sSUNI4x622_MASK_ISR *pmask)`

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Clearing Device Interrupt Enable Mask: suni4x622ClrMask

This function clears individual interrupt bits and registers in the S/UNI-4x622 device. Any bits that are set in the passed structure are cleared in the associated S/UNI-4x622 registers.

Prototype `INT4 suni4x622ClrMask(sSUNI4x622_HNDL deviceHandle, sSUNI4x622_MASK_ISR *pmask)`

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Getting SOH Interrupt Enable Mask: suni4x622GetMaskSOH

This function returns the contents of the SOH interrupt mask registers of the S/UNI-4x622 device.

Prototype `INT4 suni4x622GetMaskSOH(sSUNI4x622_HNDL deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR *pmask)`

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number

	pmask	: (pointer to) mask structure
Outputs	pmask	: (pointer to) updated mask structure

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Setting SOH Interrupt Enable Mask: suni4x622SetMaskSOH

This function sets the contents of the SOH interrupt mask registers of the S/UNI-4x622 device. Any bits that are set in the passed structure are set in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622SetMaskSOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs	deviceHandle	: device handle (from suni4x622Add)
	channel	: channel number
	pmask	: (pointer to) mask structure
Outputs	None	

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects May change the operation of the ISR / DPR

Clearing SOH Interrupt Enable Mask: suni4x622ClrMaskSOH

This function clears SOH individual interrupt bits and registers in the S/UNI-4x622 device. Any bits that are set in the passed structure are cleared in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622ClrMaskSOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs	deviceHandle	: device handle (from suni4x622Add)
	channel	: channel number
	pmask	: (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Getting LOH Interrupt Enable Mask: suni4x622GetMaskLOH

This function returns the contents of the LOH interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622GetMaskLOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs

`pmask`	: (pointer to) updated mask structure

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Setting LOH Interrupt Enable Mask: suni4x622SetMaskLOH

This function sets the contents of the LOH interrupt mask registers of the S/UNI-4x622 device. Any bits that are set in the passed structure are set in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622SetMaskLOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Clearing LOH Interrupt Enable Mask: suni4x622ClrMaskLOH

This function clears LOH individual interrupt bits and registers in the S/UNI-4x622 device. Any bits that are set in the passed structure are cleared in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622ClrMaskLOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Getting RPOH Interrupt Enable Mask: suni4x622GetMaskRPOH

This function returns the contents of the RPOH interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622GetMaskRPOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs `pmask` : (pointer to) updated mask structure

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Setting RPOH Interrupt Enable Mask: suni4x622SetMaskRPOH

This function sets the contents of the RPOH interrupt mask registers of the S/UNI-4x622 device. Any bits that are set in the passed structure are set in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622SetMaskRPOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Clearing RPOH Interrupt Enable Mask: suni4x622ClrMaskRPOH

This function clears RPOH individual interrupt bits and registers in the S/UNI-4x622 device. Any bits that are set in the passed structure are cleared in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622ClrMaskRPOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Getting PYLD Interrupt Enable Mask: suni4x622GetMaskPYLD

This function returns the contents of the PYLD interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622GetMaskPYLD(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs

`pmask`	: (pointer to) updated mask structure

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Setting PYLD Interrupt Enable Mask: suni4x622SetMaskPYLD

This function sets the contents of the PYLD interrupt mask registers of the S/UNI-4x622 device. Any bits that are set in the passed structure are set in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622SetMaskPYLD(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Clearing PYLD Interrupt Enable Mask: suni4x622ClrMaskPYLD

This function clears PYLD individual interrupt bits and registers in the S/UNI-4x622 device. Any bits that are set in the passed structure are cleared in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622ClrMaskPYLD(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Getting FIFO Interrupt Enable Mask: suni4x622GetMaskFIFO

This function returns the contents of the FIFO interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622GetMaskFIFO(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs `pmask` : (pointer to) updated mask structure

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Setting FIFO Interrupt Enable Mask: suni4x622SetMaskFIFO

This function sets the contents of the FIFO interrupt mask registers of the S/UNI-4x622 device. Any bits that are set in the passed structure are set in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622SetMaskFIFO(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Clearing FIFO Interrupt Enable Mask: suni4x622ClrMaskFIFO

This function clears FIFO individual interrupt bits and registers in the S/UNI-4x622 device. Any bits that are set in the passed structure are cleared in the associated S/UNI-4x622 registers.

Prototype
```
INT4 suni4x622ClrMaskFIFO(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Getting Line Interface Interrupt Enable Mask: suni4x622GetMaskIntfLine

This function returns the contents of the per-channel line interface interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622GetMaskIntfLine(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs

`pmask`	: (pointer to) updated mask structure

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Setting Line Interface Interrupt Enable Mask: suni4x622SetMaskIntfLine

This function sets the contents of the per-channel line interface interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622SetMaskIntfLine(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pmask`	: (pointer to) mask structure

Outputs `pmask` : (pointer to) updated mask structure

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Clearing Line Interface Interrupt Enable Mask: suni4x622ClrMaskIntfLine

This function clears the contents of the per-channel line interface interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622ClrMaskIntfLine(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`pmask` : (pointer to) mask structure

Outputs `pmask` : (pointer to) updated mask structure

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Getting System Interface Interrupt Enable Mask: suni4x622GetMaskSysIntf

This function returns the contents of the system interface interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622GetMaskSysIntf(sSUNI4x622_HNDL
deviceHandle, sSUNI4x622_MASK_ISR *pmask)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`pmask` : (pointer to) mask structure

Outputs `pmask` : (pointer to) updated mask structure

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Setting System Interface Interrupt Enable Mask: suni4x622SetMaskSysIntf

This function sets the contents of the system interface interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622SetMaskSysIntf(sSUNI4x622_HNDL
deviceHandle, sSUNI4x622_MASK_ISR *pmask)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Clearing System Interface Interrupt Enable Mask: suni4x622ClrMaskSysIntf

This function clears systerm interface interrupt bits and registers in the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622ClrMaskSysIntf(sSUNI4x622_HNDL
deviceHandle, sSUNI4x622_MASK_ISR *pmask)
```

Inputs `deviceHandle` : device handle (from `suni4x622Add`)
`pmask` : (pointer to) mask structure

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects May change the operation of the ISR / DPR

Getting APS Interrupt Enable Mask: suni4x622GetMaskAPS

This function returns the contents of the APS interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622GetMaskAPS(sSUNI4x622_HNDL
deviceHandle, UINT1 apslink, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`apslink`	: APS link number
`pmask`	: (pointer to) mask structure

Outputs

`pmask`	: (pointer to) updated mask structure

Returns

Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_APSLINK`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Setting APS Interrupt Enable Mask: suni4x622SetMaskAPS

This function sets the contents of the APS interrupt mask registers of the S/UNI-4x622 device.

Prototype
```
INT4 suni4x622SetMaskAPS(sSUNI4x622_HNDL
deviceHandle, UINT1 apslink, sSUNI4x622_MASK_ISR
*pmask)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`apslink`	: APS link number
`pmask`	: (pointer to) mask structure

Outputs

`pmask`	: (pointer to) updated mask structure

Returns

Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_APSLINK`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Clearing APS Interrupt Enable Mask: suni4x622ClrMaskAPS

This function clears the contents of the APS interrupt mask registers of the S/UNI-4x622 device.

Prototype `INT4 suni4x622ClrMaskAPS(sSUNI4x622_HNDL deviceHandle, UINT1 apslink, sSUNI4x622_MASK_ISR *pmask)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`apslink`	: APS link number
`pmask`	: (pointer to) mask structure

Outputs

`pmask`	: (pointer to) updated mask structure

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_APSLINK`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Polling the Interrupt Status Registers: suni4x622Poll

This function commands the driver to poll the interrupt registers in the device. The call will fail unless the device was initialized (via `suni4x622Init`) or configured (via `suni4x622ISRConfig`) into polling mode.

Prototype `INT4 suni4x622Poll(sSUNI4x622_HNDL deviceHandle)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_MODE`
`SUNI4x622_FAILURE`

Valid States `SUNI4x622_ACTIVE`

Side Effects None

Pseudocode
Begin
if device is configured in polling mode
call `suni4x622ISR`
End

Interrupt-Service Routine: suni4x622ISR

This function reads the state of the interrupt registers in the S/UNI-4x622 and stores them in an ISV. Performs whatever functions are needed to clear the interrupt, from simply clearing bits to complex functions. This routine is called by the application code, from within `sysSuni4x622ISRHandler`. If ISR mode is configured all interrupts that were detected are disabled and the ISV is returned to the application. Note that the application is then responsible for sending this buffer to the DPR task. If polling mode is selected, no ISV is returned to the application and the DPR is called directly with the ISV.

Prototype `void * suni4x622ISR(sSUNI4x622_HNDL deviceHandle)`

Inputs `deviceHandle` : device handle (from `suni4x622Add`)

Outputs None

Returns (pointer to) ISV buffer (to send to the DPR) or NULL (pointer)

Valid States `SUNI4x622_ACTIVE`

Side Effects None

Pseudocode
```
Begin
get an ISV buffer
update ISV with current interrupt status
if no valid interrupt condition
        return NULL
if in ISR mode
disable all detected interrupts
return ISV
else (Polling mode)
        call suni4x622DPR
        output NULL
End
```

Deferred-Processing Routine: suni4x622DPR

This function acts on data contained in the passed ISV, allocates one or more DPV buffers (via `sysSuni4x622DPVBufferGet`) and invokes one or more callbacks (if defined and enabled). This routine is called by the application code, within `sysSuni4x622DPRTask`. Note that the callbacks are responsible for releasing the passed DPV. It is recommended that it be done as soon as possible to avoid running out of DPV buffers.

Prototype `void suni4x622DPR(void *ptmpisv)`

Inputs `ptmpisv` : (pointer to) ISV buffer

Outputs None

Returns None

Valid States SUNI4x622_ACTIVE

Side Effects None

Pseudocode
```
Begin
for each ISV element (section)
get and fill out a DPV buffer
if callback (from suni4x622Init) is not NULL
invoke (section) callback
release ISV by calling sysSuni4x622ISVBufferRtn
End
```

4.12 Alarm, Status and Counts Functions

Getting the Device Status: suni4x622GetStatusChan

This function reports the current SOH, LOH, RPOH, payload and line interface status for a specific channel.

Prototype
```
INT4 suni4x622GetStatusChan(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_STATUS_CHAN
*pstatusChan)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number
pstatusChan	: (pointer to) channel status block

Outputs

pstatusChan	: (pointer to) updated channel status block

Returns
Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Getting the Device Status: suni4x622GetStatusSOH

This function retrieves the SOH status.

Prototype
```
INT4 suni4x622GetStatusSOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_STATUS_SOH
*pstatusSOH)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number

	`pstatusSOH`	: (pointer to) SOH status block
Outputs	`pstatusSOH`	: (pointer to) updated SOH status block

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Getting the Device Status: suni4x622GetStatusLOH

This function retrieves the LOH status.

Prototype
```
INT4 suni4x622GetStatusLOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_STATUS_LOH
*pstatusLOH)
```

Inputs	`deviceHandle`	: device handle (from `suni4x622Add`)
	`channel`	: channel number
	`pstatusLOH`	: (pointer to) LOH status block
Outputs	`pstatusLOH`	: (pointer to) updated LOH status block

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Getting the Device Status: suni4x622GetStatusRPOH

This function retrieves the RPOH status.

Prototype
```
INT4 suni4x622GetStatusRPOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_STATUS_RPOH
*pstatusRPOH)
```

Inputs	`deviceHandle`	: device handle (from `suni4x622Add`)
	`channel`	: channel number
	`pstatusRPOH`	: (pointer to) RPOH status block
Outputs	`pstatusRPOH`	: (pointer to) updated RPOH status block

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Getting the Device Status: suni4x622GetStatusIntfLine

This function retrieves the Line Interface status.

Prototype
```
INT4 suni4x622GetStatusIntfLine(sSUNI4x622_HNDL
deviceHandle, UINT1 channel,
sSUNI4x622_STATUS_INTF_LINE *pstatusLine)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number
pstatusLine	: (pointer to) LINE status block

Outputs

pstatusLine	: (pointer to) updated LINE status block

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Getting the Device Status: suni4x622GetStatusPYLD

This function retrieves the payload status.

Prototype
```
INT4 suni4x622GetStatusPYLD(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_STATUS_PYLD
*pstatusPYLD)
```

Inputs

deviceHandle	: device handle (from suni4x622Add)
channel	: channel number
pstatusPYLD	: (pointer to) PYLD status block

Outputs

pstatusPYLD	: (pointer to) updated PYLD status block

Returns Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV

```
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG
```

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Getting the Device Counts: suni4x622GetCountsChan

This function retrieves all the counts for a specific channel.

Prototype
```
INT4 suni4x622GetCountsChan(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_CNTR_CHAN
*pcountsChan)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pcountsChan`	: (pointer to) counter block

Outputs

`pcountsChan`	: (pointer to) updated counter block

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

Getting the Device Counts: suni4x622GetCountsSOH

This function retrieves all the SOH counts.

Prototype
```
INT4 suni4x622GetCountsSOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_CNTR_SOH
*pcountsSOH)
```

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`pcountsSOH`	: (pointer to) SOH counter block

Outputs

`pcountsSOH`	: (pointer to) updated SOH counter block

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Getting the Device Counts: suni4x622GetCountsLOH

This function retrieves all the LOH counts.

Prototype
```
INT4 suni4x622GetCountsLOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_CNTR_LOH
*pcountsLOH)
```

Inputs
- deviceHandle : device handle (from suni4x622Add)
- channel : channel number
- pcountsLOH : (pointer to) LOH counter block

Outputs
- pcountsLOH : (pointer to) updated LOH counter block

Returns
Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Getting the Device Counts: suni4x622GetCountsRPOH

This function retrieves all the RPOH counts.

Prototype
```
INT4 suni4x622GetCountsRPOH(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_CNTR_RPOH
*pcountsRPOH)
```

Inputs
- deviceHandle : device handle (from suni4x622Add)
- channel : channel number
- pcountsRPOH : (pointer to) RPOH counter block

Outputs
- pcountsRPOH : (pointer to) updated RPOH counter block

Returns
Success = SUNI4x622_SUCCESS
Failure = SUNI4x622_ERR_INVALID_DEVICE_STATE
SUNI4x622_ERR_INVALID_DEV
SUNI4x622_ERR_INVALID_CHAN
SUNI4x622_ERR_INVALID_ARG

Valid States SUNI4x622_ACTIVE, SUNI4x622_INACTIVE

Side Effects None

Getting the Device Counts: suni4x622GetCountsPYLD

This function retrieves all the PYLD counts.

Prototype
```
INT4 suni4x622GetCountsPYLD(sSUNI4x622_HNDL
deviceHandle, UINT1 channel, sSUNI4x622_CNTR_PYLD
*pcountsPYLD)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`pcountsPYLD` : (pointer to) PYLD counter block

Outputs `pcountsPYLD` : (pointer to) updated PYLD counter block

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`
`SUNI4x622_ERR_INVALID_ARG`

Valid States `SUNI4x622_ACTIVE, SUNI4x622_INACTIVE`

Side Effects None

4.13 Device Diagnostics

Testing Register Accesses: suni4x622DiagTestReg

This function verifies the hardware access to the device registers by writing and reading back values.

Prototype
```
INT4 suni4x622DiagTestReg(sSUNI4x622_HNDL
deviceHandle)
```

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)

Outputs None

Returns
Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`

Valid States `SUNI4x622_PRESENT, SUNI4x622_INACTIVE`

Side Effects None

Enabling Line Loopbacks: suni4x622DiagLineLoop

This function clears / sets a Line Loopback. It is up to the USER to perform any tests on the looped data.

Prototype `INT4 suni4x622DiagLineLoop(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`enable`	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE`

Side Effects Will inhibit the flow of active data

Enabling Path Diagnostic Loopbacks: suni4x622DiagPathLoop

This function clears / sets a Path Diagnostic Loopback. It is up to the USER to perform any tests on the looped data.

Prototype `INT4 suni4x622DiagPathLoop(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs

`deviceHandle`	: device handle (from `suni4x622Add`)
`channel`	: channel number
`enable`	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE`

Side Effects Will inhibit the flow of active data

Enabling Data Diagnostic Loopbacks: suni4x622DiagDataLoop

This function clears / sets a Data Diagnostic Loopback. It is up to the USER to perform any tests on the looped data.

Prototype `INT4 suni4x622DiagDataLoop(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`enable` : sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE`

Side Effects Will inhibit the flow of active data

Enabling Parallel Diagnostics Loopbacks: suni4x622DiagParaLoop

This function clears / sets a Parallel Diagnostics Loopback. It is up to the USER to perform any tests on the looped data.

Prototype `INT4 suni4x622DiagParaLoop(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)
`channel` : channel number
`enable` : sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE`

Side Effects Will inhibit the flow of active data

Enabling Serial Diagnostics Loopbacks: suni4x622DiagSerialLoop

This function clears / sets a Serial Diagnostics Loopback. It is up to the USER to perform any tests on the looped data.

Prototype `INT4 suni4x622DiagSerialLoop(sSUNI4x622_HNDL deviceHandle, UINT1 channel, UINT2 enable)`

Inputs
`deviceHandle` : device handle (from `suni4x622Add`)

	`channel`	: channel number
	`enable`	: sets loop if non-zero, else clears loop

Outputs None

Returns Success = `SUNI4x622_SUCCESS`
Failure = `SUNI4x622_ERR_INVALID_DEVICE_STATE`
`SUNI4x622_ERR_INVALID_DEV`
`SUNI4x622_ERR_INVALID_CHAN`

Valid States `SUNI4x622_ACTIVE`

Side Effects Will inhibit the flow of active data

4.14 Callback Functions

The S/UNI-4x622 driver has the capability to callback to functions within the USER code when certain events occur. These events and their associated callback routine declarations are detailed below. There is no USER code action that is required by the driver for these callbacks – the USER is free to implement these callbacks in any manner or else they can be deleted from the driver.

The names given to the callback functions are given as examples only. The addresses of the callback functions invoked by the `suni4x622DPR` function are passed during the `suni4x622Init` call (inside a DIV). However the USER shall use the exact same prototype. The application is left responsible for releasing the passed DPV as soon as possible (to avoid running out of DPV buffers) by calling `sysSuni4x622DPVBufferRtn` either within the callback function or later inside the application code.

Notifying the Application of SOH Events: cbackSuni4x622SOH

This callback function is provided by the USER and is used by the DPR to report significant SOH section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's addresses are passed to the driver doing the `suni4x622Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype
```
void cbackSuni4x622SOH(sSUNI4x622_USR_CTXT usrCtxt,
sSUNI4x622_DPV *pdpv)
```

Inputs	`usrCtxt`	: user context (from `suni4x622Add`)
	`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `SUNI4x622_ACTIVE`

Side Effects None

Notifying the Application of LOH Events: cbackSuni4x622LOH

This callback function is provided by the USER and is used by the DPR to report significant LOH section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's addresses are passed to the driver doing the `suni4x622Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype
```
void cbackSuni4x622LOH(sSUNI4x622_USR_CTXT usrCtxt,
sSUNI4x622_DPV *pdpv)
```

Inputs

`usrCtxt`	: user context (from `suni4x622Add`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `SUNI4x622_ACTIVE`

Side Effects None

Notifying the Application of RPOH Events: cbackSuni4x622RPOH

This callback function is provided by the USER and is used by the DPR to report significant RPOH section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's addresses are passed to the driver doing the `suni4x622Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype
```
void cbackSuni4x622RPOH(sSUNI4x622_USR_CTXT usrCtxt,
sSUNI4x622_DPV *pdpv)
```

Inputs

`usrCtxt`	: user context (from `suni4x622Add`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `SUNI4x622_ACTIVE`

Side Effects None

Notifying the Application of PYLD Events: cbackSuni4x622PYLD

This callback function is provided by the USER and is used by the DPR to report significant PYLD section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's addresses are passed to the driver doing the `suni4x622Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype
```
void cbackSuni4x622PYLD(sSUNI4x622_USR_CTXT usrCtxt,
sSUNI4x622_DPV *pdpv)
```

Inputs

`usrCtxt`	: user context (from `suni4x622Add`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `SUNI4x622_ACTIVE`

Side Effects None

Notifying the Application of SYSINTF Events: cbackSuni4x622SysIntf

This callback function is provided by the USER and is used by the DPR to report significant System Interface section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's addresses are passed to the driver doing the `suni4x622Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype
```
void cbackSuni4x622SysIntf(sSUNI4x622_USR_CTXT
usrCtxt, sSUNI4x622_DPV *pdpv)
```

Inputs

`usrCtxt`	: user context (from `suni4x622Add`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `SUNI4x622_ACTIVE`

Side Effects None

Notifying the Application of FIFO Events: cbackSuni4x622FIFO

This callback function is provided by the USER and is used by the DPR to report significant FIFO section events back to the application. This function should be non-blocking. Typically, the callback routine sends a message to another task with the event identifier and other context information. The task that receives this message can then process this information according to the system requirements. Note: the callback function's addresses are passed to the driver doing the `suni4x622Init` call. If the address of the callback function was passed as a NULL at initialization no callback will be made.

Prototype `void cbackSuni4x622FIFO(sSUNI4x622_USR_CTXT usrCtxt, sSUNI4x622_DPV *pdpv)`

Inputs

`usrCtxt`	: user context (from `suni4x622Add`)
`pdpv`	: (pointer to) DPV that describes this event

Outputs None

Returns None

Valid States `SUNI4x622_ACTIVE`

Side Effects None

5 HARDWARE INTERFACE

The S/UNI-4x622 driver interfaces directly with the USER's hardware. In this section, a listing of each point of interface is shown, along with a declaration and any specific porting instructions. It is the responsibility of the USER to connect these requirements into the hardware, either by defining a macro or by writing a function for each item listed. Care should be taken when matching parameters and return values.

5.1 Device I/O

Reading from a Device Register: sysSuni4x622Read

This function is the most basic hardware connection. It reads the contents of a specific register location. This macro should be UINT1 oriented and should be defined by the user to reflect the target system's addressing logic. There is no need for error recovery in this function.

Format `#define sysSuni4x622Read(ba, offset)`

Prototype `UINT1 sysSuni4x622Read(void * ba, UINT2 offset)`

Inputs

`ba`	: base address
`offset`	: offset from the base address

Outputs None

Returns value read from the addressed register location

Writing to a Device Register: sysSuni4x622Write

This function is the most basic hardware connection. It writes the supplied value to the specific register location. This macro should be UINT1 oriented and should be defined by the user to reflect the target system's addressing logic. There is no need for error recovery in this function.

Format `#define sysSuni4x622Write(ba, offset, data)`

Prototype
```
UINT1 sysSuni4x622Write(void * ba, UINT2 offset,
UINT1 data)
```

Inputs

`ba`	: base address
`offset`	: offset from the base address
`data`	: data to be written

Outputs None

Returns Value written to the addressed register location

Polling a Bit: sysSuni4x622PollBit

This function simply polls a register masked data until it is zero or times out.

Format `#define sysSuni4x622PollBit (base, offset, mask)`

Prototype `INT4 sysSuni4x622PollBit (void *base, UINT2 offset, UINT1 mask)`

Inputs

`base`	: base address of device being accessed
`offset`	: offset to the memory location as it appears in the hardware data-sheet.
`mask`	: mask to apply to byte read

Outputs None

Returns Success = 0
Failure = <any other value>

5.2 System-Specific Interrupt Servicing

The porting of interrupt servicing routines and tasks across platforms is a complex task. There are many different implementations of these hardware specific functions. In this driver, the user is responsible for:

- Writing an interrupt handler (`sysSuni4x622ISRHandler`) that will be installed in the interrupt vector table of the system processor. This handler shall call `suni4x622ISR` for each device that has interrupt servicing enabled to perform the Interrupt-Service Routine (ISR) related activity required by each device
- Writing the routine that installs the ISR handler in the vector table of the system processor and spawns the DPR task (`sysSuni4x622ISRHandlerInstall`)
- `Writing` the routine that removes the ISR Handler from the vector table of the system processor and deletes/suspends the DPR task (`sysSuni4x622ISRHandlerRemove`)

Installing the ISR Handler: sysSuni4x622ISRHandlerInstall

This function installs the USER-supplied Interrupt-Service Routine (ISR), `sysSuni4x622ISRHandler`, into the processor's interrupt vector table.

Format `#define sysSuni4x622ISRHandlerInstall()`

Prototype `INT4 sysSuni4x622ISRHandlerInstall(void)`

Inputs `None`

Outputs None

Returns Success = 0

Failure = <any other value>

Pseudocode
Begin
install `sysSuni4x622ISRHandler` in processor's interrupt vector table
End

ISR Handler: sysSuni4x622ISRHandler

This routine is invoked when one or more S/UNI-4x622 devices raise the interrupt line to the microprocessor. This routine invokes the driver-provided routine, `suni4x622ISR`, for each device registered with the driver.

Format `#define sysSuni4x622ISRHandler(intId)`

Prototype `Void sysSuni4x622ISRHandler(UINT4 intId)`

Inputs `intId` : interrupt identifier

Outputs None

Returns None

Pseudocode
Begin
for each device registered with the driver
  if ISR mode == ISR_MODE call `suni4x622ISR`
if returned ISV buffer is not NULL
send ISV buffer to the DPR
End

DPR Task: sysSuni4x622DPRTask

This routine is installed as a separate task within the RTOS. It runs periodically and retrieves the interrupt status information sent to it by `suni4x622ISR` and then invokes `suni4x622DPR` for the appropriate device.

Format `#define sysSuni4x622DPRTask()`

Prototype `Void sysSuni4x622DPRTask(void)`

Inputs None

Outputs None

Returns None

Pseudocode
Begin
do
wait for an ISV buffer (sent by `suni4x622ISR`)
call `suni4x622DPR` with that ISV
  loop forever

End

Removing the ISR Handler: sysSuni4x622ISRHandlerRemove

This function disables Interrupt processing for this device. Removes the USER-supplied Interrupt-Service Routine (ISR), `sysSuni4x622ISRHandler`, from the processor's interrupt vector table.

Format `#define sysSuni4x622ISRHandlerRemove()`

Prototype `Void sysSuni4x622ISRHandlerRemove(void)`

Inputs None

Outputs None

Returns None

Pseudocode
Begin
remove `sysSuni4x622ISRHandler` from the processor's interrupt vector table
End

6 RTOS INTERFACE

The S/UNI-4x622 driver requires the use of some RTOS resources. In this section, a listing of each required resource is shown, along with a declaration and any specific porting instructions. It is the responsibility of the USER to connect these requirements into the RTOS, either by defining a macro or writing a function for each item listed. Care should be taken when matching parameters and return values.

6.1 Memory Allocation / De-Allocation

Allocating Memory: sysSuni4x622MemAlloc

This function allocates specified number of bytes of memory.

Format `#define sysSuni4x622MemAlloc(numBytes)`

Prototype `UINT1 *sysSuni4x622MemAlloc(UINT4 numBytes)`

Inputs `numBytes` : number of bytes to be allocated

Outputs None

Returns Success = Pointer to first byte of allocated memory
Failure = NULL pointer (memory allocation failed)

Initialize Memory: sysSuni4x622MemSet

This function initialize a block of memory with a given value.

Format `#define sysSuni4x622MemSet(pmem, val, sz)`

Prototype
```
void sysSuni4x622MemSet(UINT1 *pmem, UINT1 val, UINT2
sz)
```

Inputs
`pmem` : (pointer to) the first byte of the memory
`val` : value to set
`sz` : size

Outputs None

Returns None

Copy Memory: sysSuni4x622MemCpy

This function copies a block of memory.

Format `#define sysSuni4x622MemSet(pdst, psrc, sz)`

Prototype `void sysSuni4x622MemSet(UINT1 *pdst, UINT1 *psrc, UINT2 sz)`

Inputs
`pdst` : (pointer to) the destination memory
`psrc` : (pointer to) the source memory
`sz` : size

Outputs None

Returns None

Freeing Memory: sysSuni4x622MemFree

This function frees memory allocated using `sysSuni4x622MemAlloc`.

Format `#define sysSuni4x622MemFree(pfirstByte)`

Prototype `void sysSuni4x622MemFree(UINT1 *pfirstByte)`

Inputs `pfirstByte` : (pointer to) first byte of the memory region being de-allocated

Outputs None

Returns None

6.2 Buffer Management

All operating systems provide some sort of buffer system, particularly for use in sending and receiving messages. The following calls, provided by the USER, allow the driver to Get and Return buffers from the RTOS. It is the USER's responsibility to create any special resources or pools to handle buffers of these sizes during the `sysSuni4x622BufferStart` call.

Starting Buffer Management: sysSuni4x622BufferStart

This function alerts the RTOS that the time has come to make sure ISV buffers and DPV buffers are available and sized correctly. This may involve the creation of new buffer pools and it may involve nothing, depending on the RTOS.

Format `#define sysSuni4x622BufferStart()`

Prototype `INT4 sysSuni4x622BufferStart(void)`

Inputs None

Outputs None

Returns Success = 0

Failure = <any other value>

Getting an ISV Buffer: sysSuni4x622ISVBufferGet

This function gets a buffer from the RTOS that will be used by the ISR code to create an Interrupt-Service Vector (ISV). The ISV consists of data transferred from the devices interrupt status registers.

Format `#define sysSuni4x622ISVBufferGet()`

Prototype `sSUNI4x622_ISV *sysSuni4x622ISVBufferGet(void)`

Inputs None

Outputs None

Returns Success = (pointer to) a ISV buffer
Failure = NULL (pointer)

Returning an ISV Buffer: sysSuni4x622ISVBufferRtn

This function returns an ISV buffer to the RTOS when the information in the block is no longer needed by the DPR.

Format `#define sysSuni4x622ISVBufferRtn(pISV)`

Prototype `void sysSuni4x622ISVBufferRtn(sSUNI4x622_ISV *pISV)`

Inputs `pISV` : (pointer to) a ISV buffer

Outputs None

Returns None

Getting a DPV Buffer: sysSuni4x622DPVBufferGet

This function gets a buffer from the RTOS that will be used by the DPR code to create a Deferred-Processing Vector (DPV). The DPV consists of information about the state of the device that is to be passed to the USER via a callback function.

Format `#define sysSuni4x622DPVBufferGet()`

Prototype `sSUNI4x622_DPV *sysSuni4x622DPVBufferGet(void)`

Inputs None

Outputs None

Returns Success = (pointer to) a DPV buffer
Failure = NULL (pointer)

Returning a DPV Buffer: sysSuni4x622DPVBufferRtn

This function returns a DPV buffer to the RTOS when the information in the block is no longer needed by the DPR.

Format `#define sysSuni4x622DPVBufferRtn(pDPV)`

Prototype `Void sysSuni4x622DPVBufferRtn(sSUNI4x622_DPV *pDPV)`

Inputs `pDPV` : (pointer to) a DPV buffer

Outputs None

Returns None

Stopping Buffer Management: sysSuni4x622BufferStop

This function alerts the RTOS that the driver no longer needs any of the ISV buffers or DPV buffers and that if any special resources were created to handle these buffers, they can be deleted now.

Format `#define sysSuni4x622BufferStop()`

Prototype `Void sysSuni4x622BufferStop(void)`

Inputs None

Outputs None

Returns None

6.3 Timers

Sleeping a Task: sysSuni4x622TimerSleep

This function suspends execution of a driver task for a specified number of milliseconds.

Format `#define sysSuni4x622TimerSleep(time)`

Prototype `Void sysSuni4x622TimerSleep(UINT4 time)`

Inputs `time` : sleep time in milliseconds

Outputs None

Returns Success = 0
Failure = <any other value>

6.4 Preemption

Disabling Preemption: sysSuni4x622PreemptDisable

This routine prevents the calling task from being preempted. If the driver is in interrupt mode, this routine locks out all interrupts as well as other tasks in the system. If the driver is in polling mode, this routine locks out other tasks only.

Format `#define sysSuni4x622PreemptDisable()`

Prototype `INT4 sysSuni4x622PreemptDisable(void)`

Inputs None

Outputs None

Returns Preemption key (passed back as an argument in `sysSuni4x622PreemptEn`)

Re-Enabling Preemption: sysSuni4x622PreemptEnable

This routine allows the calling task to be preempted. If the driver is in interrupt mode, this routine unlocks all interrupts and other tasks in the system. If the driver is in polling mode, this routine unlocks other tasks only.

Format `#define sysSuni4x622PreemptEnable(key)`

Prototype `Void sysSuni4x622PreemptEnable(INT4 key)`

Inputs `key` : preemption key (returned by `sysSuni4x622PreemptDis`)

Outputs None

Returns None

7 PORTING THE S/UNI-4X622 DRIVER

This section outlines how to port the S/UNI-4x622 device driver to your hardware and OS platform. However, this manual can offer only guidelines for porting the S/UNI-4x622 driver because each platform and application is unique.

7.1 Driver Source Files

The C source files listed in the next table contain the code for the S/UNI-4x622 driver. You may need to modify the code or develop additional code. The code is in the form of constants, macros, and functions. For the ease of porting, the code is grouped into source files (`src`) and include files (`inc`). The `src` files contain the functions and the `inc` files contain the constants and macros.

7.2 Driver Porting Procedures

The following procedures summarize how to port the S/UNI-4x622 driver to your platform. The subsequent sections describe these procedures in more detail.

To port the S/UNI-4x622 driver to your platform:

Procedure 1: Port the driver's OS extensions

Procedure 2: Port the driver to your hardware platform

Procedure 3: Port the driver's application-specific elements

Procedure 4: Build the driver

Porting Assumptions

The following porting assumptions have been made:

- It is assumed that RAM assigned to the driver's static variables is initialized to ZERO before any driver function is called.
- It is assumed that a RAM stack of 4K is available to all of the driver's non-ISR functions and that a RAM stack of 1K is available to the driver's ISR functions.
- It is assumed that there is no memory management or MMU in the system or that all accesses by the driver, to memory or hardware can be direct.

Procedure 1: Porting Driver OS Extensions

The OS extensions encapsulate all OS specific services and data types used by the driver. The `suni4x622_rtos.h` file contains data types and compiler-specific data-type definitions. It also contains macros for OS specific services used by the OS extensions. These OS extensions include:

- Task management
- Message queues
- Events
- Memory Management

In addition, you may need to modify functions that use OS specific services, such as utility and interrupt-event handling functions. The `suni4x622_rtos.c` file contains the utility and interrupt-event handler functions that use OS specific services.

To port the driver's OS extensions:

1. Modify the data types in `suni4x622_rtos.h`. The number after the type identifies the data-type size. For example, UINT4 defines a 4-byte (32-bit) unsigned integer. Substitute the compiler types that yield the desired types as defined in this file.

2. Modify the OS specific services in `suni4x622_rtos.h`. Redefine the following macros to the corresponding system calls that your target system supports:

Service Type	Macro Name	Description
Memory	`sysSuni4x622MemAlloc`	Allocates the memory block
	`sysSuni4x622MemFree`	Frees the memory block
	`sysSuni4x622MemSet`	Set the memory value
	`sysSuni4x622MemCpy`	Copies the memory block from `src` to `dest`

3. Modify the utilities and interrupt services that use OS specific services in the `suni4x622_rtos.c` and the `suni4x622_hw.c`. These file contains the utility and interrupt-event handler functions that use OS specific services. Refer to the function headers in this file for a detailed description of each of the functions listed below:

Service Type	Function Name	Description
Timer	`sysSuni4x622TimerSleep`	Sets the task execution delay in milliseconds

Interrupt	`sysSuni4x622ISRHandlerIntInstall`	Installs the interrupt handler for the OS
	`sysSuni4x622ISRHandlerRemove`	Removes the interrupt handler from the OS
	`sysSuni4x622ISRHandler`	Interrupt handler for the S/UNI-4x622 device
	`sysSuni4x622DPRTask`	Deferred interrupt-processing routine (DPR)

Procedure 2: Porting Drivers to Hardware Platforms

This section describes how to modify the driver for your hardware platform.

To port the driver to your hardware platform:

1. Modify the low-level device read/write macros in the `suni4x622_hw.h` file. You may need to modify the raw read/write access macros (`sysSuni4x622ReadReg` and `sysSuni4x622WriteReg`) to reflect your system's addressing logic.

2. Define the hardware system-configuration constants in the `suni4x622_hw.h` file. Modify the following constants to reflect your system's hardware configuration:

Device Constant	Description	Default
`SUNI4x622_MAX_DEVS`	The maximum number of S/UNI-4x622 devices on each card	5

Procedure 3: Porting Driver Application-Specific Elements

Application specific elements are configuration constants used by the API for developing an application. This section describes how to modify the application specific elements in the driver.

To port the driver's application-specific elements:

1. Define the following driver task-related constants for your OS-specific services in file `suni4x622_hw.h` and `suni4x622_rtos.h`:

Task Constant	Description	Default
`SUNI4x622_DPR_TASK_PRIORITY`	Deferred Task (DPR) task priority	85

SUNI4x622_DPR_TASK_STACK_SZ	DPR task stack size, in bytes	8192
SUNI4x622_POLL_DELAY	The constant used in polling task mode defines the interval time in millisecond between each polling action	100
SUNI4x622_TASK_SHUTDOWN_DELAY	Delay time in millisecond. When clearing the DPR loop active flag in the DPR task, this delay is used to gracefully shutdown the DPR task before deleting it	100
SUNI4x622_MAX_MSGS	The queue message depth of the queue used for pass interrupt context between the ISR task and DPR task	1000

2. Code the callback functions according to your application. The driver will call these callback functions when an event occurs on the device. The application is responsible for releasing the DPV buffer using `sysSuni4x622DPVBufferRtn` after necessary processing is completed. These functions must conform to the following prototypes:

```
void cbackSuni4x622XX(sSUNI4x622_USR_CTXT usrCtxt, sSUNI4x622_DPV
*pdpv)
```

Procedure 4: Building the Driver

This section describes how to build the driver.

To build the driver:

1. Ensure that the directory variable names in the `makefile` reflect your actual driver and directory names.

2. Compile the source files and build the driver using your make utility.

3. Link the driver to your application code.

APPENDIX A: CODING CONVENTIONS

This section describes the coding conventions used in the implementation of all PMC-Sierra driver software.

Variable Type Definitions

Table 38: Variable Type Definitions

Type	Description
`UINT1`	unsigned integer – 1 byte
`UINT2`	unsigned integer – 2 bytes
`UINT4`	unsigned integer – 4 bytes
`INT1`	signed integer – 1 byte
`INT2`	signed integer – 2 bytes
`INT4`	signed integer – 4 bytes

Naming Conventions

Table 39 presents a summary of the naming conventions followed by all PMC-Sierra driver software. A detailed description is then given in the following sub-sections.

The names used in the drivers are verbose enough to make their purpose fairly clear. This makes the code more readable. Generally, the device's name or abbreviation appears in prefix.

Table 39: Naming Conventions

Type	Case	Naming convention	Examples
Macros	Uppercase	prefix with `"m"` and device abbreviation	`mSUNI4x622_GET_BIT`
Constants	Uppercase	prefix with device abbreviation	`SUNI4x622_MAX_POLL`
Structures	Hungarian Notation	prefix with `"s"` and device abbreviation	`sSUNI4x622_DDB`
API	Hungarian	prefix with device name	`suni4x622Add()`

Type	Case	Naming convention	Examples
Functions	Notation		
Porting Functions	Hungarian Notation	prefix with "sys" and device name	sysSuni4x622Read()
Other Functions	Hungarian Notation		utilSuni4x622ValidateChan()
Variables	Hungarian Notation		maxDevs
Pointers to variables	Hungarian Notation	prefix variable name with "p"	pmaxDevs
Global variables	Hungarian Notation	prefix with device name	suni4x622Mdb

Macros

- Macro names must be all uppercase
- Words shall be separated by an underscore
- The letter 'm' in lowercase is used as a prefix to specify that it is a macro, then the device abbreviation must appear
- Example: mSUNI4x622_GET_BIT is a valid name for a macro

Constants

- Constant names must be all uppercase
- Words shall be separated by an underscore
- The device abbreviation must appear as a prefix
- Example: SUNI4x622_MAX_POLL is a valid name for a constant

Structures

- Structure names must be all uppercase
- Words shall be separated by an underscore
- The letter 's' in lowercase must be used as a prefix to specify that it is a structure, then the device abbreviation must appear
- Example: sSUNI4x622_DDB is a valid name for a structure

Functions

API Functions

- Naming of the API functions must follow the hungarian notation
- The device's full name in all lowercase shall be used as a prefix
- Example: `suni4x622Add()` is a valid name for an API function

Porting Functions

Porting functions correspond to all function that are HW and/or RTOS dependent

- Naming of the porting functions must follow the hungarian notation
- The `'sys'` prefix shall be used to indicate a porting function
- The device's name starting with an uppercase must follow the prefix
- Example: `sysSuni4x622Read()` is a hardware / RTOS specific

Other Functions

- Other functions are all the remaining functions that are part of the driver and have no special naming convention. However, they must follow the hungarian notation
- Example: `utilSuni4x622ValidateChan()` is a valid name for such a function

Variables

- Naming of variables must follow the hungarian notation
- A pointer to a variable shall use `'p'` as a prefix followed by the variable name unchanged. If the variable name already starts with a `'p'`, the first letter of the variable name may be capitalized, but this is not a requirement. Double pointers might be prefixed with `'pp'`, but this is not required
- Global variables must be identified with the device's name in all lowercase as a prefix
- Examples: `maxDevs` is a valid name for a variable, `pmaxDevs` is a valid name for a pointer to `maxDevs`, and `suni4x622Mdb` is a valid name for a global variable. Note that both `pprevBuf` and `pPrevBuf` are accepted names for a pointer to the `prevBuf` variable, and that both `pmatrix` and `ppmatrix` are accepted names for a double pointer to the variable `matrix`

File Organization

The next table presents a summary of the file naming conventions. All file names must start with the device abbreviation, followed by an underscore and the actual file name. File names convey their purpose with a minimum number of characters.

Table 44: File Naming Conventions

API (Module and Device Management)	`suni4x622_api.c`	Generic driver API block, contains Module & Device Management API such as installing/de-installing driver instances, read/writes, and initialization profiles.
API (ISR)	`suni4x622_isr.c`	Interrupt processing is handled by this block. This includes both ISR and DPR management
API (Diagnostics)	`suni4x622_diag.c`	Device diagnostic functions
API (Interface Configuration)	`suni4x622_intf.c`	Interface configuration functions for connecting the device to external interfaces (i.e., PHY, PL3, UL2, SBI, Telecombus, clk/data, LVDS, etc.)
API (Status and counts)	`suni4x622_stats.c`	Data collection block for all device results/counts not monitored through interrupt processing.
API (Device specific blocks)	`suni4x622_aps.c`, `suni4x622_loh.c`, `suni4x622_poh.c`, `suni4x622_pyld.c`, `suni4x622_soh.c`	Device specific configuration functions defined in the driver architecture.
Hardware Dependent	`suni4x622_hw.c`, `suni4x622_hw.h`	Hardware specific functions, constants and macros
RTOS Dependent	`suni4x622_rtos.c`, `suni4x622_rtos.h`	RTOS specific functions, constants and macros
Other	`suni4x622_util.c`	Utility functions
Header file	`suni4x622_api.h`	Prototypes for all the API functions of the driver
Header file	`suni4x622_err.h`	Return codes
Header file	`suni4x622_defs.h`	Constants and macros, registers and bitmaps, enumerated types
Header file	`suni4x622_typs.h`	Standard types definition (i.e., UINT1, UINT2, etc.)
Header file	`suni4x622_fns.h`	Prototypes for all the non-API functions used in the driver
Header file	`suni4x622_strs.h`	Structures definitions

APPENDIX B: ERROR CODES

The following describes the error codes used in the S/UNI 4x622 device driver

SUNI4x622_SUCCESS	Success
SUNI4x622_FAILURE	Failure
SUNI4x622_ERR_MEM_ALLOC	Memory allocation failure
SUNI4x622_ERR_INVALID_ARG	Invalid argument
SUNI4x622_ERR_INVALID_CHAN	Invalid channel number
SUNI4x622_ERR_INVALID_MODULE_STATE	Invalid module state
SUNI4x622_ERR_INVALID_MIV	Invalid Module Initialization Vector
SUNI4x622_ERR_PROFILES_FULL	Maximum number of profiles already added
SUNI4x622_ERR_INVALID_PROFILE	Invalid profile
SUNI4x622_ERR_INVALID_PROFILE_NUM	Invalid profile number
SUNI4x622_ERR_INVALID_DEVICE_STATE	Invalid device state
SUNI4x622_ERR_DEVS_FULL	Maximum number of devices already added
SUNI4x622_ERR_DEV_ALREADY_ADDED	Device already added
SUNI4x622_ERR_INVALID_DEV	Invalid device handle
SUNI4x622_ERR_INVALID_DIV	Invalid Device Initialization Vector
SUNI4x622_ERR_INT_INSTALL	Error while installing interrupts
SUNI4x622_ERR_INVALID_MODE	Invalid ISR/polling mode
SUNI4x622_ERR_INVALID_REG	Invalid register number
SUNI4x622_ERR_POLL_TIMEOUT	Time-out while polling

APPENDIX C: S/UNI-4X622 EVENTS

Section Overhead Events (SOH)

Error Code	Description
SUNI4x622_EVENT_SOH_OOF	Out Of Frame event
SUNI4x622_EVENT_SOH_LOF	Loss Of Frame event
SUNI4x622_EVENT_SOH_LOS	Loss Of Signal event
SUNI4x622_EVENT_SOH_SBIPE	Section BIP error event
SUNI4x622_EVENT_SOH_TIU	Section Trace Unstable event
SUNI4x622_EVENT_SOH_TIM	Section Trace Mismatch event

Line Overhead Events (LOH)

Error Code	Description
SUNI4x622_EVENT_LOH_LAIS	Line Alarm Signal event
SUNI4x622_EVENT_LOH_LRDI	Line Remote Defect event
SUNI4x622_EVENT_LOH_COAPS	Change of APS bytes event
SUNI4x622_EVENT_LOH_COZ1S1	Change of synchronization status event
SUNI4x622_EVENT_LOH_LBIPE	Line BIP error
SUNI4x622_EVENT_LOH_LREIE	Line REI error
SUNI4x622_EVENT_LOH_PSBF	APS Byte Failure
SUNI4x622_EVENT_LOH_SDBER	Signal Defect event
SUNI4x622_EVENT_LOH_SFBER	Signal Failure event

Path Overhead Events (RPOH)

Error Code	Description
SUNI4x622_EVENT_RPOH_TIU	Path Trace Unstable event
SUNI4x622_EVENT_RPOH_TIM	Path Trace Mismatch event
SUNI4x622_EVENT_RPOH_PSLMI	Path Signal Label Mismatch event
SUNI4x622_EVENT_RPOH_PSLUI	Path Signal Label Unstable event
SUNI4x622_EVENT_RPOH_PRDI	Path Remote Defect Indication event
SUNI4x622_EVENT_RPOH_PERDI	Path Enhanced Remote Defect Indication event
SUNI4x622_EVENT_RPOH_PBIPE	Path BIP-8 error event
SUNI4x622_EVENT_RPOH_PREI	Path REI error event
SUNI4x622_EVENT_RPOH_PAIS	Path AIS event
SUNI4x622_EVENT_RPOH_PPSE	Positive Pointer Justification event
SUNI4x622_EVENT_RPOH_PNSE	Negative Pointer Justification event
SUNI4x622_EVENT_RPOH_PLOPTR	Path Loss Of Pointer event
SUNI4x622_EVENT_RPOH_ARDI	Aux PRDI state event
SUNI4x622_EVENT_RPOH_UNEQ	Trace Identifier Equipped state event
SUNI4x622_EVENT_RPOH_PSL	Path Signal Label changed event
SUNI4x622_EVENT_RPOH_AISC	Pointer AIS event
SUNI4x622_EVENT_RPOH_LOPC	Loss of Pointer Change event
SUNI4x622_EVENT_RPOH_NEWPTR	New pointer received event
SUNI4x622_EVENT_RPOH_ILLJREQ	Illegal Pointer Justification event
SUNI4x622_EVENT_RPOH_DISCOPA	Discontinuous pointer change event
SUNI4x622_EVENT_RPOH_INVNDF	Invalid NDF event
SUNI4x622_EVENT_RPOH_ILLPTR	Illegal pointer event
SUNI4x622_EVENT_RPOH_NDF	NDF event

Payload Events (PYLD)

Error Code	Description
SUNI4x622_EVENT_PYLD_LCD	Loss of Cell Delineation event
SUNI4x622_EVENT_PYLD_CHCS	Corrected/Uncorrected HCS error event
SUNI4x622_EVENT_PYLD_FCS	FCS error event
SUNI4x622_EVENT_PYLD_RXCPXFER	Transfer of Rx CP accumulated counter data event
SUNI4x622_EVENT_PYLD_TXCPXFER	Transfer of Tx CP accumulated counter event
SUNI4x622_EVENT_PYLD_ABRT	Aborted packet received event
SUNI4x622_EVENT_PYLD_MAXL	Maximum packet length violated event
SUNI4x622_EVENT_PYLD_MINL	Minimum packet length violated event
SUNI4x622_EVENT_FIFO_RXCPFOVR	Rx Cell FIFO overrun event
SUNI4x622_EVENT_FIFO_RXFPFOVR	Rx Frame FIFO overrun event
SUNI4x622_EVENT_FIFO_TXFPFUDR	Tx Frame FIFO overrun event
SUNI4x622_EVENT_PYLD_OOCD	Change in Cell Delineation state event

Line Interface Events (INTF_LINE)

Error Code	Description
SUNI4x622_EVENT_INTF_LINE_WANS	WANS phase detector averaging period has begun event
SUNI4x622_EVENT_INTF_LINE_LOT	Loss of Transition event
SUNI4x622_EVENT_INTF_LINE_ROOL	Recovered Reference out of lock event
SUNI4x622_EVENT_INTF_LINE_DOOL	Recovered data out of lock event

System Interface Events (SYS_INTF)

Error Code	Description
SUNI4x622_EVENT_SYS_INTF_TXOP	event that occurs when TSOP or TSEP is not asserted with the first or last word of a POS-PHY

Error Code	Description
	packet
SUNI4x622_EVENT_SYS_INTF_UNPROV	event that occurs when a non-existent channel buffer is detected during in-band addressing
SUNI4x622_EVENT_SYS_INTF_CAM	data field mismatch event
SUNI4x622_EVENT_SYS_INTF_TPRTY	Tx Parity error event
SUNI4x622_EVENT_SYS_INTF_TSOC	start of cell re-alignment interrupt
SUNI4x622_EVENT_SYS_INTF_FOVR	FIFO overrun event
SUNI4x622_EVENT_SYS_INTF_FUNR	FIFO underrun event

Automatic Protection Switching Events (APS)

Error Code	Description
SUNI4x622_EVENT_APS_BIP	BIP-8 error event
SUNI4x622_EVENT_APS_LOS	Loss of Signal event
SUNI4x622_EVENT_APS_LOF	Loss of Frame
SUNI4x622_EVENT_APS_OOF	Out of Frame
SUNI4x622_EVENT_APS_LOT	Loss of Transition
SUNI4x622_EVENT_APS_DOOL	Recovered Data out of lock
SUNI4x622_EVENT_APS_ROOL	Recovered reference out of lock
SUNI4x622_EVENT_APS_ESE	Elastic store FIFO error
SUNI4x622_EVENT_APS_PJ	Pointer Justification

LIST OF TERMS

APPLICATION: Refers to protocol software used in a real system as well as validation software written to validate the S/UNI-4x622 driver on a validation platform.

API (Application Programming Interface): Describes the connection between this module and the user's application code.

ISR (Interrupt-Service Routine): A common function for intercepting and servicing device events. This function is kept as short as possible because an Interrupt preempts every other function starting the moment it occurs and gives the service function the highest priority while running. Data is collected, Interrupt indicators are cleared and the function ended.

DPR (Deferred-Processing Routine): This function is installed as a task, at a user configurable priority, that serves as the next logical step in Interrupt processing. Data that was collected by the ISR is analyzed and then calls are made into the application that inform it of the events that caused the ISR in the first place. Because this function is operating at the task level, the user can decide on its importance in the system, relative to other functions.

DEVICE: ONE S/UNI-4x622 Integrated Circuit. There can be many devices, all served by this one driver module.

- DIV (Device Initialization Vector): Structure passed from the API to the device during initialization; it contains parameters that identify the specific modes and arrangements of the physical device being initialized.
- DDB (Device Data Block): Structure that holds the Configuration Data for each device.
- DSB (Device Status Block): Structure that holds the Alarms, Status, and Counts for each device.

MODULE: All of the code that is part of this driver, there is only one instance of this module connected to one or more S/UNI-4x622 chips.

- MIV (Module Initialization Vector): Structure passed from the API to the module during initialization, it contains parameters that identify the specific characteristics of the driver module being initialized.
- MDB (Module Data Block): Structure that holds the Configuration Data for this module.

RTOS (Real Time Operating System): The host for this driver.

ACRONYMS

API: Application Programming Interface

APS: Automatic Protection Switch

ATM: Asynchronous Transfer Mode

DDB: Device Data Block

DIV: Device Initialization Vector

DPR: Deferred-Processing Routine

DPV: Deferred-Processing (routine) Vector

DSB: Device Status Block

FIFO: First In, First Out

ISR: Interrupt-Service Routine

ISV: Interrupt-Service (routine) Vector

LOH: Line Overhead

MDB: Module Data Block

MIV: Module Initialization Vector

POH: Path Overhead

PYLD: Payload

RPOH: Receive Path Overhead

RTOS: Real-Time Operating System

SDH: Synchronous Digital Hierarchy

SONET: Synchronous Optical Network

TPOH: Transmit Path Overhead

INDEX

A

api functions

C

callbacks

constants

D

device

driver

system-specific functions